Coordinating services embedded everywhere via hierarchical planning

Georgievski, Ilche

Coordinating services embedded everywhere via hierarchical planning

Ilche Georgievski

Supported by the Netherlands Organisation for Scientific Research (NWO) under contract number 647.000.004 within the scope of the Smart Energy Systems program.

ISBN: 978-90-367-8148-0 (book)
ISBN: 978-90-367-8147-3 (e-book)

Printed by *NetzoDruk* - www.netzodruk.nl - Groningen

Cover drawing by Ana & Borche Georgievski. The cover is a rendition of an intelligent system, which appears as a human head with a network holding and processing information in hierarchical forms.

rijksuniversiteit
groningen

Coordinating services embedded everywhere via hierarchical planning

Proefschrift

ter verkrijging van de graad van doctor aan de
Rijksuniversiteit Groningen
op gezag van de
rector magnificus prof. dr. E. Sterken
en volgens besluit van het College voor Promoties.

De openbare verdediging zal plaatsvinden op

vrijdag 9 oktober 2015 om 11.00 uur

door

Ilche Georgievski

geboren op 30 juli 1986
te Bitola, Macedonië

Promotor
Prof. dr. M. Aiello

Beoordelingscommissie
Prof. dr. C. Bettini
Prof. dr. H. G. Sol
Prof. dr. A. Tate

To Vangjel and Snezhana

Contents

Acknowledgements

Reminiscing about the period when I joined the Distributed systems group back in December 2010 until the present, I found myself smiling in satisfaction. I know this is because of the contributions, assistance, and care of many people I have encountered during this period of my life. To all of you, my gratitude is great.

Prof. dr. Marco Aiello made this experience of mine possible in the first place by taking me in as an intern and later on as a PhD student. Marco, you are one of the most considerate people I have ever met. Thank you for your excellent guidance and generous support during this whole period. Probably of most benefit to me, you taught me to be critical and strive for excellence in science. I can only now see how your advice help me grew proficient. Your open-door policy made a convenient way of supervision, making you quickly a voice of reason in my PhD. You gave me freedom of work as no one would hope for, and you patiently and with encouragement guided my work into the current shape. I will always treasure the knowledge and habits acquired from you. I am truly honoured to have you as my *promotor*.

Concerning the members of the reading committee, Prof. dr. Claudio Bettini, Prof. dr. Henk G. Sol, and Prof. dr. Austin Tate, I thank you for the time and effort you put into reading and reviewing my thesis. Prof. dr. Austin Tate showed a proactive and attentive stance toward my work on HTN planning, and gave me feedback on Nonlin and O-Plan2, for which I am deeply grateful. I am thankful for the insights concerning graphical representations and conceptual models that Prof. dr. Henk G. Sol relayed on me.

Two other academics gave me their time and insight to develop and improve my work. I am grateful to Prof. dr. Luigia Carlucci Aiello for being attentive to details on my work in Chapter 4. I thank Prof. dr. Luis Castillo Vidal for the openness and feedback on SIADEX.

To the ladies at the secretariat, Esmee, Ineke, Desiree and Helga, you were not only helpful in handling all paperwork and resolving practical issues, but you also made the environment more joyful with your stories and laughter. Thank you for all that.

I shared unique, scientific and personal experiences with my current and former colleagues: Eirini, Tuan, Ehsan, Viktoriya, Faris, Heerko, Fatimah, Frank, Ang, Ando, Brian, Azkario, Pavel, Doina, Saleem, Mahir, Kerstin, George, and Giannis. Some special thanks in order. Eirini, thank you for infecting me with AI planning, and the feedback on my work in Chapter 4. A thanks goes to the co-authors with whom I have published papers in appreciation of sharing their knowledge and putting effort to create something useful to all of us: Viktoriya, Tuan, Andrea, Faris, Brian, Alexander, and Marco. Alexander, I am also thankful for making me use new technologies in as shortest time as possible. Those technologies now realise an important part of this thesis. Though sometimes it was not easy to get something I needed from you, we still had several helpful discussions. Faris and Tuan, thank you for considering me to be part of your Sustainable Buildings story. Best of luck with creating a fascinating story! Fatimah, I was always delighted by your sincerity in our conversations during lunch. A special thanks goes to Frank for translating my lay abstract into Dutch! I thank Pieter Noordhuis for the initial implementation of the **SH** planner.

I especially want to thank my colleagues that have been my office mates over the years. Ehsan, in addition to the conversations about our countries and politics, I tasted delicious Pakistani food and had many moments of laughter thanks to you. Heerko, we had nice chats about films and series. I cannot but thank you that my "free" time was dedicated to a new series almost always on your suggestion. I am deeply grateful for the Dutch translation of the thesis summary. Tuan, thanks for the agreeable conversations, the music we listen to together, the Vietnamese food you shared with me. We have been office mates for the longest period, which is quite some time to listen to my disappointments and ramblings. You deserved to be my paranymph!

I have been fortunate to have met wonderful people in Groningen. Violeta and Marija, I will be always grateful for all the attention, nice food, support and care you have given to me. On top of this, we were an excellent dancing trio! Violeta, our chats and loud кафе-муабети were the joining of two good friends to escape from whatever was going on and to laugh. Therefore, you join my defence as my paranymph! Marija, your dedication to people amazes me. I must also thank you for all the drama you have brought in my life! Ena, the food, the laughter, the understanding we shared. Thanks for all the nice memories. Bibi and Viktor, we became good friends on a fast track. Thank you for your kindness and hospitality. Mina,

thank you for your uplifting presence! Funda, I feel like I know you for long time. Thank you for your cheerful spirit and Turkish experiences. A special note to Vlad for being my cinema and beer buddy. Saba, thank you for the engaging conversations, and watering my flowers while I was away. Finally, to everyone not personally mentioned but help me to keep my peace and made my time in Groningen enjoyable, I am thankful, wholeheartedly.

To my friends in Macedonia, Maja, Marija, Andrijana, Ana (P) and Aleksandra: I am grateful for your support and attention. Elena (Ѓ), we had exciting chats about life, culture, and beyond. Thank you for the wonderful memories from my stay in Belgium. Elena (B), we grew up together and we still share similar experiences. Thank you for having me in Germany. Goran, despite you being in Greece, we still found time and places to share thoughts on existential questions. Thank you for your interest in my PhD! Anita, I am grateful for your devotion to me despite your continuous cruising around the world. Viktor, you were updated daily about whatever was happening in my life. You must have great patience and understanding. Thank you! Tanja, you believed the most in me. I reminisce the past and I cannot tell you how an extraordinary friend you are. Thank you for every moment you spent on me! Ana (M), very special gratitude to you. Your effervescent personality has a profound effect on me. Thank you for being empathetic and supportive all these years!

I am grateful to all my relatives for the care and attention given to me. My energy and spirit were regularly renewed during holidays at home. Oh, and my suitcases were filled up too!

To my brother Borche and his wife Ana: Thank you for your love! Your tenderness and light-hearted nature help me stay upbeat. Thank you also for taking the challenge and creating the artistic drawing that has adorned the outside of this thesis.

Nothing would have been possible without my parents. It is to them that I dedicate my thesis. Тато и мама, не постојат зборови што можат да го изразат тоа што го чувствувам, но еве, ќе се обидам. Ви благодарам за сета љубов што ми да ја давате, за сета ваша пожртвуваност, што ми ја покажавте убавината на светот, и што ме исполнивте со бесконечна надеж. Ви благодарам што ме научивте сè што можевте и што ми верувавте да го научам останатото самостојно. Без вас би бил никаде и ништо.

Ilche
Groningen
1 September 2015

Chapter 1

Introduction

Computing exists and takes place all around us. All the more so nowadays, when devices are embedded and applications are present everywhere in our environments. One such environment is the new home of Theodore, a writer. While each room is equipped with various home appliances, such as a TV in the living room, his home is enriched with numerous unobtrusive devices, such as radio-frequency identification tags, temperature and gas-leakage sensors, actuators to switch lamps, *etc.* As Theodore is an early adopter, he also bought Tars, a domestic robot capable of performing diverse tasks, such as sensing human presence, cleaning rooms, moving around the home, picking up and dropping items, and helping and supporting a user. All these devices, including the robot, provide various kinds of information and means for operation using a wide range of communication and integration technologies, typically seen through the prism of services. A *service* is an abstraction of a software component from its implementation details in the form of interfaces. These interfaces are most commonly accompanied with explicit functional descriptions. Services are distributed over various types of networks, and interoperate with each other by exchanging messages. A lamp, for example, has services for both sensing and changing its state. The true benefits of such enriched and abstracted environments appear however when the focus of computing is not on the devices alone, but on their *coordination* for the purpose of greater good, such as improving people's experiences and quality of life, energy saving, or safety. Supporting the activities and serving the needs of people in unobtrusively equipped environments is otherwise commonly referenced as *ubiquitous computing* or *pervasive computing* or *ambient intelligence* (Weiser 1999).

Theodore now decides to purchase a system that will do such computing, that is, deal with his needs and requests, anticipate his activities, process the information from devices, coordinate all devices and appliances, cooperate with Tars, and take care of the home. Samantha is named the system he obtains.[1] Today, to build a system like Samantha, one would have environment (home) adaptations programmed

[1] Theodore, Samantha and Tars are inspired by the eponymous characters in the films "Her" (Phoenix and Johansson 2013) and "Interstellar" (Irwin 2014).

or defined beforehand. This means that all requests and situations that may happen in Theodore's home need to be predicted and covered in the adaptations. Designing such adaptations requires a strenuous mental and manual activity which usually results in only a limited number of objectives or situations included. For example, it is relatively easy for Samantha to instruct Tars to clean the home from dust in such a way that he does not disturb Theodore using predefined instructions. But, if we need Tars to clean the kitchen and the bedroom and to deliver some items to Theodore also, what would be the (best) instruction for Tars to accomplish this?

While specifying adaptations or instructions, one assumes that involved services are always available and executable in some ideal, or more precisely, predictable setting. All environments, including Theodore's home, are however characterised by a certain degree of uncertainty. This means that if a change in an objective, service availability or the environment itself occurs, it is most likely that some adaptations will no longer be applicable, and the system will fail to react. Let us illustrate this through a situation in Theodore's home.

One day Theodore decides to prepare lunch. While he chooses a dish from the home menu, Samantha takes into account the recipe for that dish and selects a set of instructions for Theodore to follow. During computation, Samantha finds out that an ingredient is missing that cannot be replaced with any of the available ones in the kitchen. Unfortunately, Samantha has no predefined solution to this problem. If she is intelligent, she will instruct Tars to go to the storage room and get the missing ingredient, enabling Theodore to still cook his chosen dish. In the case of emergency situations, the consequences of not knowing what to do could be worse, exemplified as follows. While Theodore is preparing the lunch, Samantha detects a gas leak, which is always a dangerous situation (Kaldeli et al. 2012). Samantha triggers a predefined goal for dealing with such situations, and computes a safety solution. It consists of instructions for Theodore to leave the home, actions to close all doors leading to the kitchen so as to isolate spreading of the gas as much as possible. At the same time, Samantha issues actions to pull up window blinds and open the window in the kitchen. However, it happens that the blinds are stuck, which prevents the window from opening. This situation has not been foreseen at the design time, and therefore Samantha has no solution for it.

With the device proliferation, modifications of existing adaptations are required, and most often, new adaptations are needed. If all these updates are performed without any systematic steps, the outcome will be a cluttered system. At the end, such a system is difficult to reuse across different types of environments, which is, if nothing more, undesirable from a business perspective.

Samantha must be able to continuously find ways to transform the current state of Theodore's home into a state that satisfies his requests or deals with some newly

arisen situation. We need techniques that exhibit autonomous and intelligent behaviour in physical environments that goes beyond the knowledge predefined by people.

1.1 Coordination via AI planning

The field of Artificial Intelligence (AI) deals with building systems that are capable of intelligent behaviour, and AI planning provides means for automated and dynamic coordination of services, for example, (Ranganathan and Campbell 2004, Sirin et al. 2004, Berardi et al. 2008). *Planning* is the process of selecting and coordinating actions by considering their outcomes in order to achieve a given objective, and *AI planning* deals with this process computationally (Ghallab et al. 2004). Henceforth, when we use the term 'planning', we refer strictly to AI planning. Considering the coordination of services, actions correspond to services and objectives to requests. In addition to automation, we gain several more benefits by using planning. First, planning supports naturally the evolution of systems as the adaptation of actions is relatively easy and flexible. Second, the provided knowledge can be modified and maintained in an organised and conceptual way – the purpose of each planning construct is always known. Third, the same knowledge with minor modifications can be suitable to a wide range of ubiquitous computing environments. Finally, planning can provide the means to maximise people's comfort and energy savings.

1.1.1 Characterisation of planning for ubiquitous computing

The basic and evident correspondence between planning and ubiquitous computing environments is exploited in several existing studies, for example, (Kotsovinos and Vukovic 2005, Amigoni et al. 2005, Kaldeli et al. 2012, Rocco et al. 2014). What appears to be less obvious is how ubiquitous computing environments are related to planning beyond device services and user requests. Through the example of Theodore's home, we see that the environments are associated with other attributes too, such as temporal and spatial relations, user preferences, human actions, uncertainty, *etc.* Looking at the existing studies, the correspondence and the extent to which planning corresponds to these attributes cannot be easily grasped. The main obstacle to recognise and interpret these issues is planted into ambiguities of planning-based approaches designed for and only little applied to ubiquitous computing environments.

One way to address these concerns is to have a view of planning for ubiquitous computing such that explains the entities constituting the field and presents the relationships between them. The main benefit of such a view would be the better

understanding of the field, independent of design and implementation concerns. A stable view can then support subsequent and facilitated development of ubiquitous computing solutions. This is necessary if the long-term objectives of ubiquitous computing reach beyond just ideas or partially functioning prototypes. To the best of our knowledge, there is no existing view that abstracts away planning for ubiquitous computing. We therefore deal with this issue and organise the aspects of planning for ubiquitous computing in a specification that can be clearly comprehended, easily communicated, and used for subsequent design and development.

Ubiquitous computing is a domain in which systems are expected to react fast. In many cases, such as the emergency situation in Theodore's home, the speed of reaction is crucial. This means that the speed of computing of all techniques involved in the system's life cycle is important for the overall system reaction. We have to be particularly aware about the performance of computationally expensive tasks, such as planning (Erol et al. 1995). In this context, little is known about how difficult it is to coordinate ubiquitous computing environments via planning, or the amount of resources such coordination requires. To begin with, one can gain some knowledge on the upper bound of speed and length of solutions produced by some planning techniques in ubiquitous computing. We can achieve this by analysing the complexity of planning in specific domains (Helmert 2003). In order to have insights into the complexity of planning for ubiquitous computing, we define a general ubiquitous computing planning domain. This enables the characterisation of two decision problems, one relating to the length of solutions for the planning problems, and the other one involving the existence of solutions.

The classical approach to planning requires an initial state of an environment, a goal state, and a set of actions (Ghallab et al. 2004). Classical planning tries to find such sequence of actions that transforms the initial state into the goal one. This approach implies at least two difficulties for ubiquitous computing environments. One is the creation of the goal state, which specifies declaratively *what* has to be achieved. This task might be easy for some specialised software, such as rule-based engines (Degeler and Lazovik 2013), but it is a real challenge for users to conceive an intended goal, one that is not in the list of predefined ones. The second difficulty lies in specifying actions. An action typically consists of *preconditions*, defining when the action can be applied, and *effects*, specifying the expected outcome of the action application. Since actions correspond to services, it means that actions would contain only simple prepositions to check whether corresponding services are executable given the current state. However, this cannot be the case, as planning techniques usually require knowledge that goes beyond such basic conditions. That is, actions need to be annotated with semantics additional to the description provided in services (Marquardt and Uhrmacher 2009a, Kaldeli et al. 2012). There are two pos-

sibilities to overcome this. One involves services to come along with appropriate semantics, meaning that this would be a responsibility deferred to manufacturers of devices. The other option would be to have a domain author or expert responsible for maintaining and updating service descriptions accordingly and whenever necessary. If, for example, a device is excluded from the environment, its services and therefore actions must be removed from the domain too. This activity is far from easy in practice, especially in more complex ubiquitous computing domains where causal relations between actions can be easily lost.

The first difficulty can be overcome by having an objective which indicates that something needs to be done or performed without specificities of what and how. We can deal with the second difficulty by keeping services without specific knowledge. It is useful when such knowledge is encoded and maintained independently from services. Fortunately, there is a planning technique that provides these and many more features for modelling and supporting ubiquitous computing environments.

1.1.2 Hierarchical task network planning

Hierarchical Task Network (HTN) planning, or *hierarchical planning*, is a planning technique that breaks with the tradition of classical planning (Sacerdoti 1975a, Tate 1977, Erol et al. 1994a). The basic idea behind this technique includes an initial state description, a task network as an objective to be accomplished, and domain knowledge consisting of networks of primitive and compound tasks. A *task network* represents a hierarchy of tasks each of which can be executed, if the task is *primitive*, or decomposed through *methods* into refined subtasks, if the task is *compound*. Planning starts by decomposing the initial task network and continues until all compound tasks are decomposed, that is, a solution is found. The solution is a *plan* which equates to a set of primitive tasks applicable to the initial world state.

HTN planning is a particularly useful technique due to its *rich* domain knowledge provided in task networks. It is generally well suited for domains in which some hierarchical representation is desirable or known in advance, domains that encourage complex and composite constructs, domains that involve structured strategies, and domains that are naturally epistemic. HTNs enable encoding information about *how* to perform some task or strategy, or how to accomplish something. This is especially convenient for services describing some environment-specific processes and strategies involving other services. In this context, HTNs allow encoding knowledge at varying levels of abstraction, focusing thus on a particular level at a time (Sirin et al. 2004). With this *modularity*, services of different granularity can be modelled as primitive and compound tasks.

Causal reasoning in more complex ubiquitous computing domains can be eas-

ily lost if only planning actions are used. Hierarchical planning through modularity can help here by allowing device services to be encoded at the bottom level of the hierarchy. These services would then constitute the next more coarse-grained level (Yordanova 2011). In addition, this *structured causality* supports and eases the evolution of ubiquitous computing environments. If, for example, the evolution means a change in the ubiquitous computing system so as to fulfil new user requirements, then the causality of HTNs helps in determining which services have been affected by the change. Once such services are identified, the *flexibility* of HTNs makes it relatively easy to plug in new services as methods, or remove the obsolete ones without drastically affecting the hierarchy.

HTN planning supports *control flow*. HTNs enable controlling the order in which services are executed, *e.g.,* partial order, or methods evaluated (in the if-then-else manner). This provides a ground for modelling composite services with several control constructs, such as sequence, unordered, choice, if-then-else, *etc.* (see (Sirin et al. 2004) for examples). In this context, HTNs support recursion too. A compound task that is applied within its own definition is called *recursive*. In most cases, recursive tasks accomplish something by dividing it into smaller parts each of which is addressed with a recursive call to the task until reaching the base case. Recursive tasks aid the modelling of services that show a propensity for cycles.

The knowledge encoded in HTNs helps in reducing the search space of planners[2] and therefore finding a solution reasonably fast, if one exists. This makes hierarchical planners a "good compromise" between wide reusability and *effectiveness* in comparison to classical, domain-independent planners (Marquardt et al. 2008). The latter planners do not require domain-specific knowledge, and thus they are widely applicable, however, they are characterised by weak efficiency. In ubiquitous computing, domains usually contain a large and constantly increasing number of services. HTN planning fits for such domains because it *scales* well to large number of tasks, and also generally to increasing size of planning problems.

The main shortcoming of using HTN planning is the lack of instructions or recommendations on how to model domain knowledge. Conceiving the knowledge depends on the capabilities of the domain author, his expertise in the ubiquitous computing domain, and his understanding of the underlying planning system and modelling language. However, this is true for all kinds of planning, not just HTN planning.

[2]We use a planner and a planning system interchangeably for the implementation of a planning technique.

1.2 A closer look at HTN planning

Though long-standing and widely used, HTN planning is characterised by controversy and lack of a common understanding (Georgievski and Aiello 2015a). This situation cannot be effortlessly clarified because the current literature on HTN planning, despite being rich, reports little or noting at all on the issues, especially in a consolidated form. We introduce a new viewpoint on HTN planning, where we differentiate between plan-based HTN and state-based HTN planning, considering the kind of space the search is performed in. Both models can achieve a more or less similar level of expressiveness, and one model would be more preferable than the other one depending on the expressivity constructs needed for the planning domain at hand. For example, one can anticipate partially ordered plans easily with planners of the former model. The main shortcoming of resorting to plan-based HTN planning is the necessity for complex and complicated problem-solving mechanisms. These include resolution methods for task interactions, management of constraints in task networks and those posted during planning, heuristics, *etc.* In addition, the field of plan-based HTN has been dormant, the state-of-the-art planners have an ambiguously defined syntax, and their underlying systems are tightly coupled (*i.e.,* their components are highly dependent on one another), making them difficult to extend and improve.

For practical reasons, we choose to work with state-based HTN planning. In contrast to plan-based HTN planning, this model requires simpler mechanisms in planners, which may provide the benefits of loosely coupled systems (*i.e.,* easy to extend and improve). Looking at the state-based HTN planners, little is known about the internal architecture of SIADEX (Castillo et al. 2006). Also, the planner is not publicly available, preventing us from inspecting its implementation characteristics. On the other hand, SHOP2 is a state-based HTN planner distributed under an open-source licence (Nau et al. 2003), whose most recent and modern version is the Java implementation JSHOP2 (Ilghami 2006). JSHOP2 uses code generation to transform an HTN planning problem specified in the SHOP2 syntax into executable code. This approach introduces some inconvenience in the manipulation and extension of the planner.

The main drawback of state-based HTN planners is their requirement for elaborate domain knowledge. In state-of-the-art planners, the domain knowledge may even include constructs that provide behaviour of programming languages, something that goes beyond the expectations for AI planners. For example, lists and operations on lists in the SHOP family of planners, or Python-based domain functions in SIADEX. In addition, the syntax of SHOP2 is 'liberal' with respect to what is allowed to appear in numerical expressions in both, preconditions and effects,

arguments of predicates, and values of parameters in tasks. In SIADEX, the syntax is mostly based on the Planning Domain Definition Language (PDDL) (McDermott et al. 1998) (to be precise, on the 2.1 version of PDDL (Fox and Long 2003)), which is a *de facto* standard language for AI planners. We refer to the hierarchical syntax based on PDDL as Hierarchical Planning Definition Language (HPDL). HPDL makes things much clearer for state-based HTN planning, as compared to SHOP2 one, though the semantics underlying the language are left undefined.

Other reasons for the well-conceived knowledge provided to state-based HTN planners can be found in the special encodings needed for achieving predicates, modelling the base cases of recursive tasks, modelling the book-keeping primitive tasks (used to track what needs to be done further during planning), *etc.* (Georgievski and Aiello 2015a). The way of handling base cases of recursive tasks is by using additional methods, which represents one aspect of phantomisation. *Phantomisation* defines what happens after the process of inferring that some task is already accomplished by some other tasks in the task network. As opposed to plan-based HTN planners, the phantomisation in state-based HTN planners is achieved explicitly using the knowledge in the domain. Whether phantomisation situations will be identiified and encoded appropriately depends directly on the abilities of the domain author.

The last issue we focus on is of technical nature and refers to the ability of planners to integrate in large software systems. To take the case of ubiquitous computing, the systems need to consider scalability (*e.g.,* with respect to devices or software components), distribution, interoperability, evolution, and reusability (Degeler et al. 2013). Most of these challenges are open issues for planners: as part of complex and distributed systems, the planners need to provide computational power on demand and to support and guarantee location and distribution transparency. This necessity arises from the complexity and diversity of planning problems in real-world domains. In the case of ubiquitous computing, factors that contribute towards complexity are: (1) the size of the planning problem which may vary with the addition of, for example, new devices in the environment, (2) the planning implementation that can reside on different locations (*e.g.,* in the home or in the Cloud), (3) the distribution of the domain knowledge which might be centralised or decentralised, and (4) the decision-making control which might lead to conflicts between AI-based and user-based decisions. As for diversity, planning problems may involve a wide range of scenarios, from homes, office buildings, to hospitals.

We set out to develop a new HTN planning system, one with a higher degree of simplicity and flexibility than state-of-the-art state-based HTN planners. We call it *Scalable Hierarchical (**SH**) planner*. We use HPDL for the planner because, first, it paves the way to have a unified and clear representation for state-based HTN

planners, and second, we want to have clearly defined syntax for numerical expressions. Numeric-state variables and numerical expressions are convenient constructs for ubiquitous computing environments (Kaldeli et al. 2012). In this context, we formally extend state-based HTN planning to include numerical expressions. Under this extension, we define a state to be a pair of sets of predicates and numerical variables. The values of variables are updated through the application of primitive tasks. So, we treat numbers only as values associated with objects, and not as unique and independent objects in the environment (in the manner of PDDL (Fox and Long 2003)). We further discharge authors from the responsibility to identify and encode phantomisation situations and provide a way to perform phantomisation automatically. Finally, we propose the concept of *planning as a service* following the principles of Service-Oriented Computing (SOC) (Papazoglou and Georgakopoulos 2003, Erl 2007). This means that planning functionalities are offered as services in order to support easy and efficient integration and development of cooperative systems.

1.3 Establishing relationships with ubiquitous computing

Many situations in real-world environments, including ubiquitous computing ones, may produce multiple alternative ways of accomplishing some objective, where each such alternative is associated with some risk. Risk presents the possibility to win or loose some wealth or resource, such as money, energy, time, satisfaction, and sometimes humans (*e.g.*, in emergency situations). The sensitivity to risk can be analysed through utility theory, which deals with decision making in accordance to some risk attitude of people (Neumann and Morgenstern 1947). For example, for a risk-averse person loosing part of some resource means more than winning a certain amount. *Utility* then expresses the preference over choices. When making decisions, people or systems try to enhance their utility. If Theodore expresses a preference for natural light over artificial light, it can be reasoned that Theodore's utility will be higher when blinds are pulled up than when they are down and lamps are turned on.

Risk sensitivity has been considered in planning models conceptually different than HTN planning, *e.g.*, (Koenig and Simmons 1994). In the field of HTN planning, there are approaches that deal only with user preferences (Sohrabi et al. 2009), costs of primitive tasks (Nau et al. 2003, Luo et al. 2013), predefined costs of compound tasks (Amigoni et al. 2005), and user ratings and trust associated with primitive and compound tasks, respectively (Kuter and Golbeck 2009). We take an approach similar to the one in (Kuter and Golbeck 2009), where user ratings are used to cal-

culate backwards the trust in compound tasks. We assume non-positive costs for primitive tasks, expressing some property of consumption, and we assign utilities to compound tasks, indicating the attitude toward the risk of consuming the given resource. Our approach, *utility-based HTN planning*, tries to find a plan that maximises some utility function given a resource.

Understanding ubiquitous computing problems clearly makes the formulation of their corresponding planning problems (or reasoning counterparts) easier and sound. The main reason for this necessity lies in the diversity and complexity of the problems in ubiquitous computing (Bettini et al. 2010). Knowing the focus and constituents of the addressed ubiquitous computing problem introduces a definite way of creating a planning problem corresponding to it. The correspondence makes a space to further concentrate on issues related exclusively to planning problems, such as specification of the corresponding planning problem at the planning level, the way of creating such specification in a sense whether it can be automatically induced from the environment or it must depend on the expertise of the domain author, and finally, the need for expressivity constructs in order to master the problem under consideration. We take this approach and model a specific problem of ubiquitous computing environments upon which we define the corresponding HTN planning problem. This provides the means to declare that the plan computed for the planning problem is indeed a solution to the initial ubiquitous computing problem.

The plans are computed *off-line*, meaning the planning state is assumed to be updated only by the steps involved in the plans. In ubiquitous computing environments this represents a strong assumption given their dynamism and uncertainty. Events, which usually represent environment changes, occur all the time during execution, unexpectedly and asynchronously. As an example, recall the missing ingredient for Theodore's dish or the window blinds getting stuck. The events affect the plan computed off-line by making some of plan steps obsolete or the entire plan invalid. Additionally, we may encounter unpredicted behaviour from services at the time of execution. A service may fail executing, it may not respond at all, or it may return a result different from the expected one (Lazovik 2006, Kaldeli 2013).

There are at least two ways to deal with these issues. One involves computing a conditional plan with branches for all possible outcomes that may occur during execution and affect the plan, for example, (Hoffmann and Brafman 2006). Given the nature of ubiquitous computing environments, this seems inadequate: it is difficult to predict the outcomes and their number is potentially high and probably increasing with the proliferation of devices. The other approach interleaves planning, monitoring and execution, *e.g.,* (Brenner and Nebel 2009, Kaldeli et al. 2011). Here, a plan computed off-line is monitored while executing its actions. In case of unex-

pected behaviour coming from changes in the environment or action executions, a planner is invoked to modify the plan. If the planner is unable to revise the plan, a new plan is computed. The main drawback of this approach is the computational burden it causes to the underlying system, especially in cases when the planner spends time and resources on revising a plan and eventually ends up making a new plan. Though the revision time can be constrained, it still affects the speed of system reaction, which is crucial for most situations in ubiquitous computing environments.

Due to these reasons, we resort to a more pragmatic approach. In face of inconsistencies, the plan execution continues only if the remainder of the plan is not affected by the environment change or service failure. Otherwise, a new plan is computed. We formalise the execution semantics using the concept of *orchestration* in the manner of SOC. Orchestration is the process of coordinating and executing services with the purpose to carry out the specified service composition (Erl 2007). Most often, the orchestration process is assigned to a central component that interacts with other components of the underlying system. Analogous to this, we use orchestration to coordinate the receipt of events, planning for new HTN planning problems, and execution of their corresponding plans. The process creates and maintains HTN planning domain and state, and upon each newly received event, it updates the state and formulates an HTN planning problem. Afterwards, it invokes the planner to solve the problem, and if plan is found, the orchestration executes it. If an event is received that affects the execution or a service execution fails, the orchestration asks the planner to compute a new plan. Otherwise, the plan execution continues.

1.4 Ubiquitous computing systems

Systems *à la* Samantha need to interact with the devices in the environment, collect and interpret the data coming from them, maintain some additional information about the environment, and take care of the coordination and execution of services. We design an architecture suitable for such systems to which we refer as Hierarchical Task Network Planning for Ubiquitous Computing (HTNPUC) architecture. HTNPUC follows the principles of service-orientation on both the ubiquitous and application levels. This means that, in addition to devices providing services, the capabilities of architecture components are offered as services too. We refer to the former ones as *ubiquitous services*, and to the latter ones as *application services*. Conceptualising everything as a service provides for a distributed, scalable and dynamic infrastructure.

Figure 1.1 depicts the components of the HTNPUC architecture and their inter-

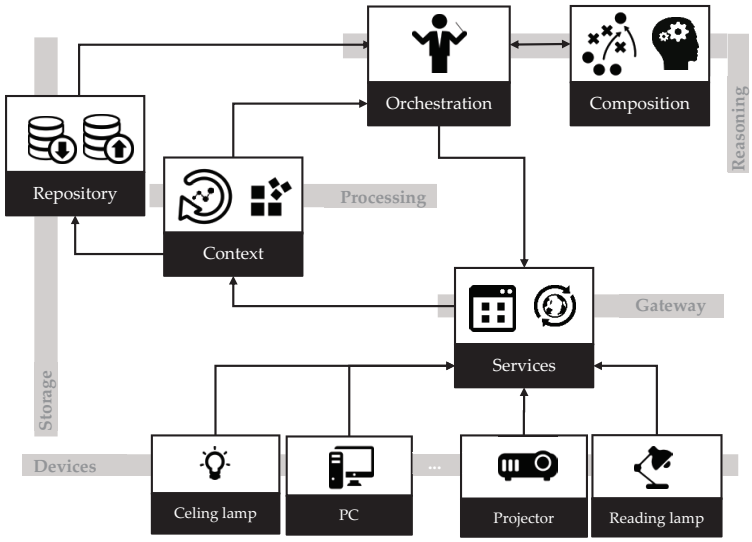


Figure 1.1: High-level overview of HTNPUC.

action. The devices are represented by sensors, actuators, and appliances. Each device has an interface that provides an access to device's data and control. The interfaces may use different protocols for communication. Devices are organised in several types of networks providing a basic infrastructure for data manipulation. The gateway is the point where a unified way of interaction with devices is provided, that is, device functionalities are encapsulated as services and offered to interested components of the architecture.

The repository component stores mainly two types of environment information. One involves descriptions of devices deployed in the environment, their types and locations within the environment, their data type and value ranges, the layout of the environment, *etc.* The other type of information is dynamic and involves the data coming periodically from devices directly or through the context component. The context component accepts data arriving from devices and continuously monitors the state of each device. This component collects data, aggregates it into meaningful information, and provides it to interested components. The last functionality is based on a publish and subscribe mechanism, enabling components to show interest about specific information and get notifications when related events occur, for example, environment changes.

Reasoning is accomplished through the use of two components, one that represents our orchestration process and the other one refers to the composition of

ubiquitous services via hierarchical planning. Since ubiquitous computing environments are expected to support continuous orchestration, we adopt a strategy that enables long-running runtime activities. Thus, the orchestration component receives events from the context component through the publish and subscribe mechanism, and reacts immediately and accordingly. The composition component, on the other hand, is represented by the **SH** planner. The component receives an HTN planning problem and computes a plan whose execution ensures that the environment is adapted according to the initial task network and environment-specific conditions.

We realise HTNPUC as a system prototype. We deploy the system and demonstrate its applicability in an actual environment. We use the facilities of our own building, Bernoulliborg, at the University of Groningen, The Netherlands. Bernoulliborg is a building of more than 10000 m^2 erected in 2008.[3] We exploit the space that is located on the ground floor and used as a restaurant between 11.30 a.m. and 2 p.m. every working day, and used for reading, working and other social activities the rest of the time. With the system, we want to see how automated coordination of lamps brings benefits related to both, energy savings, and user acceptance and satisfaction.

With this system we may achieve reduction of energy use, and as a consequence, monetary savings, in an implicit way thanks to the knowledge provided in HTNs. What appears lately to be an explicit necessity for ubiquitous computing environments is the optimisation of the price paid for energy using the information coming from the electricity system (Georgievski et al. 2012). The assumption here is that the electricity system will evolve into a *smart grid* in which dynamic prices for electricity are offered from competing providers (Pagani 2014). This means that we can acquire amounts of energy from different providers in short-time intervals, say, every hour. Optimality comes into perspective when the cheapest energy is bought whenever possible considering the amount of energy needed to coordinate devices. The devices are coordinated in a way to avoid peak tariffs as much as possible. In order to explore this option, we build and deploy another prototype of a ubiquitous computing system on the fifth floor of Bernoulliborg. The prototype involves components for monitoring devices, storing data about devices and energy providers, communicating with the smart grid, and scheduling and controlling devices. In contrast to the previous system, here the coordination of the components is performed in a centralised manner.

The main idea behind the second system is to create temporal plans optimised with respect to the amount of energy bought from the cheapest provider. We may refer to temporal planning as a way to create plans whose steps are temporally

[3] `http://nl.wikipedia.org/wiki/Bernoulliborg`

annotated (Ghallab et al. 2004). However, temporal planning methods usually focus on optimising one dimension only, the one of minimising the plan duration, *e.g.*, (Smith and Weld 1999, Bacchus and Kabanza 2000, Do and Kambhampati 2003, Gerevini et al. 2006), and more recent methods that additionally consider PDDL-based preferences (Benton et al. 2012). An alternative approach, which we adopt, uses techniques specialised for scheduling and optimisation problems (Nizamic et al. 2012).

1.5 A way to compose applications automatically

Each of these systems offers its own specific capabilities appropriate for ubiquitous computing environments. However, it may happen that a combination of the capabilities of both systems suits better user needs. The outcome is a customised system (or application) composed of components in a way that meets user requirements. We expect that the composition and purchase of such systems are possible in the framework of *cloud computing* (Hayes 2008). The system would be then present everywhere, meaning silently delivered to any infrastructure, place or device enabled with Internet. Irrespective of where the system is deployed and executed, its composition will consist of possibly distributed software components of different granularity, each offering one or more application services. The *coordination* of these services creates values in systems and involved corporations that may go beyond standard expectations, such as resource utilisation (*e.g.*, servers) and economics. The coordination of application services is usually achieved either manually, as in our cases, or with some predefined scripts. Both approaches make the coordination difficult due to the high interrelation of components, 'versioning' of components, including different requirements for communication and exchange of messages, multiplication of component instances with the increase of the size of environments, and, as a consequence, a varying number of application services.

We can address these issues by automating the process of coordinating application services. One way to achieve this is to consider application services as the ones accessible on the public Web, and use planning techniques to compose them, *e.g.*, (Aiello et al. 2002, Sirin et al. 2004, Lazovik et al. 2004, Medjahed and Bouguettaya 2005, Klusch and Gerber 2005, Kaldeli et al. 2011). The main shortcoming of using Web services is the lack of consistent semantic descriptions despite the existence of several notable specifications (*e.g.*, SOAP, WSDL, OWL-S) (Fan and Kambhampati 2005). This makes Web service composition highly infeasible in practice. Another way is to see application services as Cloud services, that is, those whose accessibility may not necessarily be public. Cloud services are usually accessible within a limited number of corporations, ensuring greater service con-

trol and privacy. Services in well-controlled environments are more structured and annotated with semantics using a consistent (in-house) ontology. Indeed, corporations are willing to use standards and best practices gained from SOC to enable a well-defined access to services. These observations anticipate the feasibility of composing Cloud services. So, we take the challenge and automate the coordination of application services using HTN planning.

1.6 Thesis scope, approach, and organisation

The research challenges that we tackle can be organised in three topics. The first topic is concerned with the issues related to the characterisation and understanding of two fields, planning for ubiquitous computing and HTN planning. The second topic comprises the issues within the scope of state-based HTN planning, and the third topic encompasses the challenges related to development and application of solutions in ubiquitous computing environments and cloud computing environments.

The approach we adopt to deal with the challenges under the first topic is based on qualitative research (Glaser and Strauss 2009, Corbin and Strauss 2008). All sciences, including Computer Science, characterise the nature of phenomenons under consideration qualitatively from which more detailed knowledge can be further developed (Newell and Simon 1976). Among qualitative methodologies, grounded theory enables developing artefacts, such as concepts, categories and a theory, through an analysis of data (Corbin and Strauss 1990). The artefacts should offer an explanation about the phenomenon under examination. In fact, the grounded theory methodology provides a guidance for data collection and procedures for data analysis.

We use the methodology of grounded theory to organise and analyse existing studies on planning for ubiquitous computing. Our contribution represents a novel overview of this field. In contrast to existing surveys on service composition in ubiquitous computing, where planning is subsumed as one approach to service composition (Urbieta et al. 2008, Stavropoulos et al. 2011, Brnsted et al. 2010), our overview deals extensively with planning only, and is performed rigorously and systematically using a wide range of factors, starting from those relevant to the environments, AI planning, to elements important for efficient demonstrations of ubiquitous computing approaches. Using the artefacts resulting from the systematisation and conceptual modelling (Mylopoulos 1992, Thalheim 2010), we introduce a conceptual model that helps in explaining and communicating planning for ubiquitous computing. In addition, using the systematisation too, we contribute theoretically by defining a general planning domain for ubiquitous computing, and laying a basis

for considering computational properties of planning problems within that domain. We accomplish this following the approach taken by Helmert (2003).

The field of HTN planning is examined qualitatively too. Using the existing planners and studies, information about formal models, concepts and properties are extracted, analysed and interpreted in a consolidated form. To the best of our knowledge, this is the first comprehensive viewpoint on HTN planning. Our main contribution lies in the categorisations of the field, and clarifications of many misconceptions associated with this planning technique.

The approach taken to address the issues under the second and third topics includes identification of a problem, development of a solution to the problem, and evaluation of the solution. This approach in fact is implicitly represented by design science research, which is concerned with devising artefacts in order to attain goals (Simon 1996, Iivari and Venable 2009). In other words, design science research aims to develop artefacts, such as models, algorithms, and tools, and evaluate them in order to ensure their usefulness within the domain of interest. So, our approach can be reflected using the design science research methodology which consists of six steps (Peffers et al. 2007). The first one involves the identification of a specific problem and the motivation for its solution. The second step includes the definition of the solution objectives. The third step involves designing new artefacts, followed by the demonstration of the artefacts in solving instances of the specified problem. The fifth step comprises the evaluation of the design in terms of efficiency, effectiveness, acceptance, *etc.* The last step concerns the communication of the problem and design to an appropriate audience.

We follow mostly all six steps when dealing with each of the challenges in the second and third topics. In the second topic, we address deficiencies, such as a lack of well-defined semantics for numerical expressions and of automatic phantomisation, identified during the analysis of the field of HTN planning. On the other hand, we also deal with problem-initiated challenges, such as the support for utilities and service-orientation. The main contributions from the design and development under this topic are the foundation of utility-based HTN planning, the concept of planning as a service, and **SH**, a new HTN planner that supports the syntax of HPDL, which is based on the well-defined semantics, and offers its capabilities as services.

In the third topic, we consider issues initiated from the context of our research problems. In the domain of ubiquitous computing, we design one solution based on HTN planning to coordinate ubiquitous services. The main contributions include an establishment of a correspondence between a ubiquitous computing environment and an HTN planning problem, execution semantics, and a demonstration of the proposed solution in an actual environment, enabling us to prove the feasibility of the solution. We construct another solution based on scheduling to control en-

vironments connected to the smart grid. Our contributions include a way to control devices considering dynamic energy prices that come from various providers, and a demonstration of the solution in another actual environment. We prove the feasibility of the proposed solution and we show the potential to monetary and energy savings. In the domain of cloud computing, we design a solution for composing application services using HTN planning. Our contributions are the formulation of the relationship between the problem of creating applications ready for deployment and an HTN planning problem, and a demonstration of the feasibility of the solution.

Following the topics, the organisation of the thesis begins with a broader perspective involving descriptions and analysis of planning for ubiquitous computing, continues narrower by dealing with HTN planning, and finally, focuses specifically on the application areas.

Chapter 2 introduces the systematisation of planning for ubiquitous computing. We provide a formal description of classical planning which is useful for the discussions throughout the thesis. We then present the methodology used to search for and analyse existing studies in this field. The core part contains descriptions and discussions organised in three sets of classes: ubiquitous computing environments, AI planning, and interpretation of ubiquitous computing approaches.

Chapter 3 presents the conceptual model for planning in ubiquitous computing. We here also analyse planning domains, scenarios and descriptions within the scope of the systematisation from Chapter 2, and define a general ubiquitous computing planning domain. We introduce initial results on the complexity of solving planning problems in this domain.

Chapter 4 revisits hierarchical planning. It contains definitions necessary for the understanding of the two models of HTN planning. We then extract and describe concepts essential to the search process of hierarchical planners, and we give an overview of the state-of-the-art planners with respect to those concepts. We also define a set of properties according to which we further analyse state-of-the-art planners.

Chapter 5 contains the features advancing state-based HTN planning. We define the numerically extended state-based HTN planning, we describe the basic concepts and an algorithm needed for automatic phantomisation, and we propose the utility-based HTN planning.

Chapter 6 provides details on the concept of planning as a service and on the **SH** planner. We first discuss the service-orientation of planning systems and we propose a few classes of planning services. We then give insights into the engineering and implementation of **SH**.

Chapter 7 goes into details of the use of HTN planning and orchestration in

ubiquitous computing environments. We show how a ubiquitous computing environment is described and what does a ubiquitous computing problem consist of. Based on this, we demonstrate the formulation of the corresponding HTN planning problem. With the orchestration and its semantics, we are able to close the life cycle of the ubiquitous computing system. We also provide details on the realisation of the system deployed in the restaurant of Bernoulliborg. We evaluate the approach with respect to energy and monetary savings, usability, and performance of **SH**.

Chapter 8 demonstrates the approach we propose for ubiquitous computing environments connected to the smart grid. We explain the model and the parameters upon which the optimisation is performed. The system architecture is discussed with an emphasis on the centralised coordination. We then detail the implementation and the results on monetary and energy savings we achieve by deploying the system to the offices on the fifth floor of Bernoulliborg.

Chapter 9 introduces our approach for automated composition of Cloud services into applications ready for deployment. We explain how the deployment model is defined and how a relationship with HTN planning can be established. We then demonstrate the applicability and feasibility of the approach on a set of experiments. Finally, we analyse Web service composition especially when performed via HTN planning as an alternative approach for composing applications. We review existing approaches and discuss the current shortcomings.

Chapter 10 concludes the thesis. We discuss the main achievements, and we present some directions for further considerations.

Chapter 2

Systematisation of planning for ubiquitous computing

Planning is generally accepted as a relevant technique to achieve goal-oriented behaviour of ubiquitous computing environments. Its acceptance is by virtue of its supposed suitability to address many issues that the environments face. These include dynamism and uncertainty, reasoning about time and parallelism of actions, distribution of devices, modelling of the domain knowledge, and so forth. In this context, there are numerous planning techniques envisioned, defined and applied to ubiquitous computing. While the matchmaking may be evident, it is not clear how planning is actually designed for, used, and integrated in ubiquitous computing environments. This is symptomatic especially because many of the proposed approaches have unclear premises and differ in too many extents. These issues relate to, first, the characteristics of the many domains in ubiquitous computing, such as homes, hospitals, and offices; second, the capabilities of planning techniques; then, the models and languages used to define the environments as planning problems; and, finally, the purpose of planning. If we exemplify the last point, one can notice that many of the scenarios for which planning is envisioned are rather simplistic. For instance, planning is regarded as appropriate to provide a step-by-step guidance when a home inhabitant performs some activity, such as to remind, let's say, Theodore, to turn off the tap when brushing his teeth in the bathroom (Simpson et al. 2006). A slightly more complex scenario involves planning in order to achieve partial or complete automation of a specific activity – the underlying system performs actions on behalf of Theodore. In other words, the system offloads tasks from Theodore because it takes responsibility for activities that are "repetitive, highly predictable, or require little judgement", and of activities related to the safety of Theodore and the home itself. For instance, Samantha may turn off the air conditioner when Theodore leaves his home. Planning is also thought as useful for identifying emergencies when the behaviour of inhabitants deteriorates and deviates from some predefined patterns.

We find exactly these issues as a reason to make a systematisation of existing studies employing planning for ubiquitous computing environments. We start by

defining what planning is. We then describe the methodology used for the systematisation of planning for ubiquitous computing. More specifically, we employ a systematic review to select existing studies, and a qualitative analysis to extract information from them. By using that information, we derive a set of classes that enable us to focus and discuss specific topics related to planning and ubiquitous computing. Finally, we provide some implications of the analysis and challenges for the research in this area.

2.1 Classical planning

Usually, planning relies on the concept of a *state model*, which is defined over a state space and associated with a single initial state, a non-empty set of goal states, and a set of actions that deterministically map each state to another (Bonet and Geffner 2000).

2.1 Definition (State model). *A state model $\mathcal{M}$ is a tuple $\langle S, s_0, S_G, A, \delta \rangle$, where*

- *S is the finite and discrete set of states,*
- *$s_0 \in S$ is the initial state,*
- *$S_G \subseteq S$ is the set of goal states,*
- *A is the finite set of actions,*
- *$\delta : S \times A \to S$ is the deterministic transition function.*

  The set of applicable actions in a state $s \in S$ is defined over all actions $a \in A$ such that (s, a) is in the domain of δ.

The application of an action a to a state s results in state $s' = \delta(s, a)$, or equivalently $s' = s[a]$. The application of a sequence of actions $a_1, \ldots, a_n$ to a state s is defined as

$$s[] = s$$
$$s[a_1, \ldots, a_2] = (s[a_1, \ldots, a_{n-1}])[a_n]$$

This state model is the one determining *classical planning*. A classical planning problem concerns finding a sequence of actions that maps a specified initial state to some goal state. The sequence of actions $a_1, \ldots, a_n$ that results in a sequence of states $s_0, \ldots, s_{n+1}$ such that a_i is applicable in s_i, its application results in $s_{i+1} = s_i[a_i]$, and $s_{n+1} \in S_G$ is called a *solution* to the classical planning problem, or a *plan*.

2.2 Definition (Plan). *Let $\mathcal{M} = \langle S, s_0, S_G, A, \delta \rangle$ be a state model. The sequence of actions $\pi = a_1, \ldots, a_n$ is a plan for $\mathcal{M}$ if and only if $s_0[\pi] \in S_G$.*

With the increase of the size and complexity of planning problems, which happens often, the state space grows exponentially, and thus, its enumeration becomes infeasible. To circumvent this, states can consist only of a set of values for variables with finite and discrete domains. The actions, their applicability and the transition function are then defined in terms of these variables.

In planning, the most common way of representing such states is based on propositional variables (also *facts*, *atoms* or *fluents*). A propositional variable has a domain of two values, *true* or *false*, which determine whether some proposition about the world holds in a given state. In fact, this is known as STRIPS representation,[1] and the corresponding problem as STRIPS planning problem (Fikes and Nilsson 1971).

2.3 Definition (STRIPS planning problem). *A STRIPS planning problem $\mathcal{P}^S$ is a tuple $\langle F, O, I, G \rangle$, where*

- *F is the set of propositional variables,*
- *O is the set of operators each of which is of the form $\langle pre(o), add(o), del(o) \rangle$, where $pre(o), add(o), del(o) \subseteq F$,*
- *$I \subseteq F$ is the initial state,*
- *$G \subseteq F$ is the goal state.*

The STRIPS planning problem captures the state model determining the classical planning problem implicitly. A state $s \in S$ is a subset of F such that variables $v \in s$ have value *true*. An assumption is that the variables $v' \in F \setminus s$ have value *false*. This is in fact the *closed-world assumption* in which all and only the propositions that are true are specified in the state. Further, I corresponds to the initial state s_0, while G to a set of goal states $S_G = \{s \mid G \subseteq s\}$. The set of applicable actions in state s corresponds to the actions whose preconditions evaluate to true, that is, $\{o \in O \mid pre(o) \subseteq s\}$. The transition function progresses a state s with operator o by adding propositions $add(o)$ to s and subtracting $del(o)$ from s, that is, $\delta(s, o) = (s \cup add(o)) \setminus del(o)$. A sequence of actions $a_1, \ldots, a_n$ is a plan π if each action a_i is applicable in s_i, that is, $pre(a_i) \subseteq s_i$, and the state resulting from the application of π from the initial state $s_0 = I$ contains the goal state G, that is, $G \subseteq s[\pi] \in S_G$.

[1] The original representation was in first-order logic, but due to technical difficulties, it was reduced to propositional logic (Nilsson 1980).

A successor of the STRIPS representation is the Planning Domain Definition Language (PDDL) (McDermott et al. 1998), nowadays a standard in planning. PDDL extends STRIPS to first-order logic with a finite sets of constants, variables and predicates. Since for the discussions in this chapter and further in the thesis a clear understanding of the STRIPS planning problem suffices, we avoid providing a formal model of a PDDL planning problem, while we refer for PDDL-related definitions to (Helmert 2009).

2.2 Methodology

We adopt a mixed approach to find and classify existing studies on planning for ubiquitous computing. It consists of (1) a comprehensive search for studies relevant to the particular subject based on a systematic method (Klassen et al. 1998, Kitchenham and Charters 2007, Petticrew and Roberts 2006), and (2) an identification of classes and classification of relevant studies based on qualitative analysis (Corbin and Strauss 2008, Glaser and Strauss 2009). While here we describe our approach briefly, we provide the whole procedure in (Georgievski and Aiello 2015b).

We reduce the potential for research bias by employing a review protocol founded upon the one provided in (Kitchenham and Charters 2007). Our modified review protocol consists of the following main steps: formulation of research questions, search for studies in two phases, and selection of studies. We formulate our research questions by using the Population, Intervention, Outcome, Context structure (Pai et al. 2004, Petticrew and Roberts 2006, Kitchenham and Charters 2007). The population is represented by ubiquitous computing environments, while the intervention is planning as a technique to achieve automation within these environments. The outcomes refer to capabilities of planning to address the points of question in ubiquitous computing environments, such as requests and user preferences, temporal aspect of automation, uncertainty and dynamism, the degree of formality of planning problems, and the characterisation of how well planning techniques operate within ubiquitous computing environments.

We perform the search for relevant studies in two phases. In the first phase, we look at published surveys on service composition in ubiquitous computing (Urbieta et al. 2008, Stavropoulos et al. 2011), and we also include several studies we knew about before starting the review process. In the second phase, we search for relevant papers in several electronic databases.

Our selection strategy has two steps, called *screens*. In the first screen, we check the titles and abstracts of all found papers, and we include a paper only if several criteria are satisfied. In this screen, we exclude non-English studies, master or doctoral theses, surveys, and proceeding reports or workshop reports. In the second

Table 2.1: List of primary studies.

ID	Study	ID	Study
S1	(Qasem et al. 2004)	S20	(Sánchez-Garzón et al. 2012)
S2	(Ranganathan and Campbell 2004)	S21	(Pajares Ferrando and Onaindia 2013)
S3	(Kotsovinos and Vukovic 2005)	S22	(Fraile et al. 2013)
S4	(Amigoni et al. 2005)	S23	(Ha et al. 2005)
S5	(Ding et al. 2006)	S24	(Krüger et al. 2011)
S6	(Vukovic et al. 2007)	S25	(Grześ et al. 2014)
S7	(Carolis and Cozzolongo 2007)	S26	(Marquardt et al. 2008)
S8	(Courtemanche et al. 2008)	S27	(Heider 2003)
S9	(Bajo et al. 2009)	S28	(Rocco et al. 2014)
S10	(Liang et al. 2010)	S29	(Cirillo et al. 2012)
S11	(Masellis et al. 2010)	S30	(Madkour et al. 2013)
S12	(Mastrogiovanni et al. 2010)	S31	(Ortiz et al. 2013)
S13	(Santofimia et al. 2010)	S32	(Milani and Poggioni 2007)
S14	(Bidot et al. 2011)	S33	(Georgievski et al. 2013)
S15	(Yordanova 2011)	S34	(Garro et al. 2008)
S16	(Sando and Hishiyama 2011)	S35	(Marquardt and Uhrmacher 2009a)
S17	(Hidalgo et al. 2011)	S36	(Jih, Hsu, Lee and Chen 2007)
S18	(Kaldeli et al. 2012)	S37	(Harrington and Cahill 2011)
S19	(Song and Lee 2013)		

screen, we read entirely the studies resulted from the first screen, and we exclude a paper if it represents a proceedings version of a journal article appearing among the selected studies.

We collect nineteen unique papers from the first phase. We find however only 40% of them relevant for this treatment. Further, we collect 1150 candidate studies from six databases in the second phase. Among these, we identify 50 relevant studies in the first screen. We perform the second screen on these 50 studies, and we exclude 26% of the papers. We therefore have a total number of 37 studies to which we refer as *primary studies* and upon which we perform the qualitative analysis. Table 2.1 shows the primary studies.

We derive a set of classes by using a class identification process founded upon the one provided in (Smidts et al. 2014) and based on qualitative analysis (Saldana 2009). The qualitative analysis enables us to review the primary studies and identify summative and silent attributes, and repeating patterns from their text. We use such attributes and patterns to segregate and organise data into classes. Once we identify classes out of the text from the primary studies, we use a classification process to systematically arrange the primary studies into these classes. Hence, the studies belonging to the same class share some characteristics, and therefore we can consolidate meaning and explanation effectively.

2.3 Classes of properties

We organise classes into three main perspectives, namely environments, planning, and interpretation. The first perspective focuses on ubiquitous computing environments, where we regard a ubiquitous computing environment as the state model defined in Definition 2.1. The information contained within a state of a ubiquitous computing environment is naturally spatial and temporal (Cook and Das 2004, Guesgen and Marsland 2010, Bettini and Riboni 2015). The spatial notion describes the relationship that a person or an object has with the space it occupies or acts upon, but it also defines the layout of the environment – the relationships between spaces and the arrangement of an individual space into locations. The temporal notion expresses simultaneity or ordering in time of events and actions.

A ubiquitous computing environment, in essence, aims at improving the experience, comfort and productivity of its inhabitants. Every person basically should be empowered to express personal preferences for, for example, events, actions and adaptations occurring in the environment. In addition to preferring specific states of the environment, a request can be either issued explicitly by a person or inferred by other software components. These preferences and requests represent goal states of the ubiquitous computing environment. Further, the adaptation of the environments is enabled by the use of actions that represent various entities, such as actuators, sensors, appliances, *etc.* These actions have the power to change the behaviour of an environment when they act collectively in it. By being dynamic state models, ubiquitous computing environments are inherently characterised by uncertainty. Uncertainty concerns all problems and contingencies occurring throughout the whole life cycle of an environment.

This first perspective gives a basis for the second one, which looks at AI planning. We identify two important issues related to planning problems. The first one concerns the modelling of planning problems in such a way that they capture ubiquitous computing environments. In this context, the modelling language and its expressive power have an important role. A modelling language enables expressing the physical properties of an environment and advice about how to handle specific situations that may occur in it. We have to be aware that any model of the physics makes simplifying assumptions and abstracts behaviours, so this is the case with the models of ubiquitous computing environments too. Given the fact that these environments are associated with a variety of properties, such as time, space, object types, preferences, *etc.*, the power of the modelling language to express constructs that encapsulate such properties is crucial.

The second issue is related to the representation of planning problems. Problem representation must reflect the model of the environment as accurately as possible,

and it also provides a computational framework. Given a model of an environment, the question that arises is how to generate the problem definition out of such a model. Moreover, how accurate will that definition be?

In addition to these issues, a point in question are the reasoning capabilities of planning with respect to the different properties of ubiquitous computing environments (space, time, preferences, *etc.*). Another concern for planning and related to ubiquitous computing is uncertainty which planning systems traditionally handle by monitoring the execution of actions and taking recovery steps to solve any potential contingency (Musliner et al. 1991, Veloso et al. 1998, Lazovik et al. 2003, Kaldeli 2013).

The third perspective unites the other two and represents the development of planning applications, their use in real ubiquitous computing environments, and evaluation of their performance in terms of responsiveness and stability under different loads. This practical perspective also encapsulates the evaluation of the level of usability of planning systems by users.

We use these three perspectives of planning for ubiquitous computing to organise the set of classes we identify in the classification process as the hierarchy shown in Figure 2.1. Each of the three top-level classes represents one of the perspectives. In the following, we discuss each class separately.

2.3.1 Environments

The class of *Environments* focuses on several dimensions of ubiquitous computing environments, such as user intentions, sources of environment changes, physics of the surrounding, and uncertainty. We therefore analyse the Environments class by dividing it in the following subclasses: *Behavioural inputs*, *Behavioural outputs*, *Physical properties*, and *Uncertainty*. The Behavioural inputs class deals with requests and preferences of the people in environments, while the Behavioural outputs class focuses on the types of actions that transform the state of an environment. The Physical properties class defines an environment with respect to the spatial and temporal dimensions. Finally, the Uncertainty class characterise the sources of contingencies in these environments. The classification of primary studies with respect to these classes is provided in Table 2.2.

Behavioural inputs classes

We define *behavioural input* as the information representing someone's desires according to which a ubiquitous computing environment should behave. The behavioural input is transformed into a convenient form for a planning system, and given as a part of the system's input. In the following, we discuss two classes of

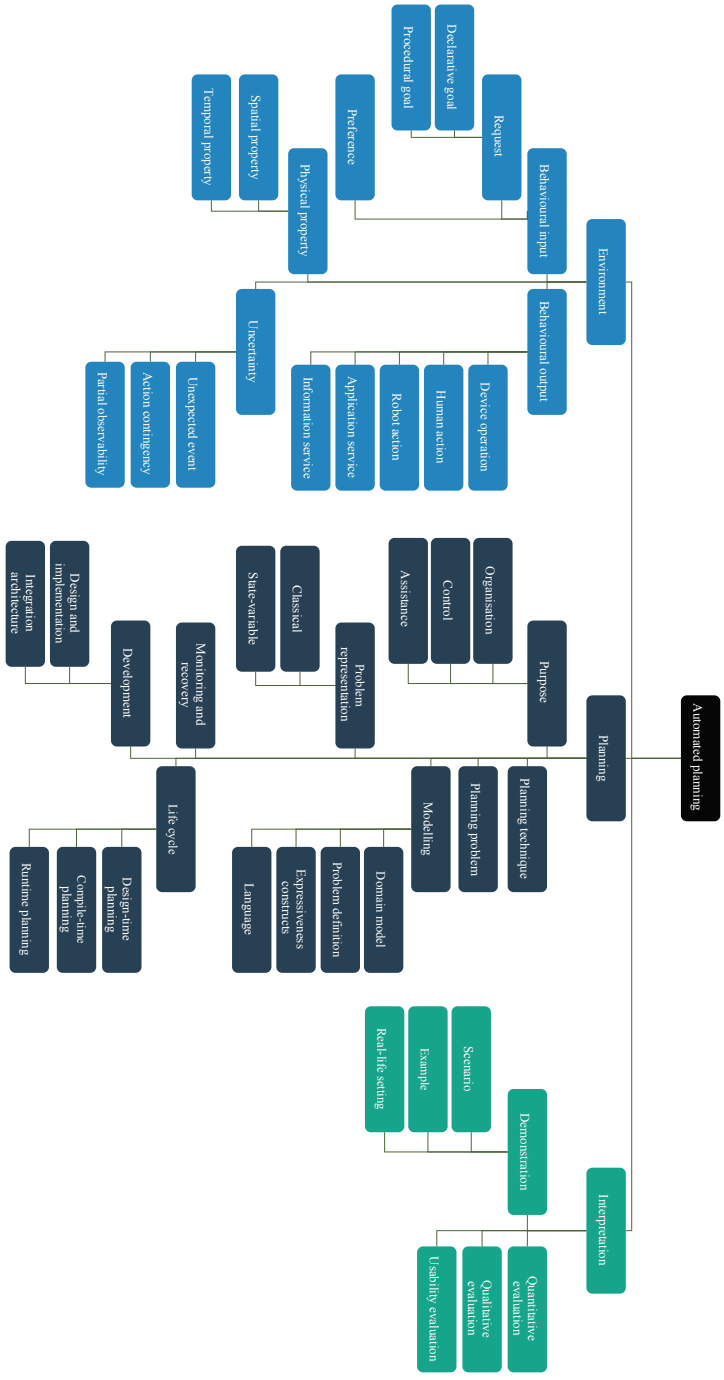


Figure 2.1: Hierarchy of classes of planning for ubiquitous computing.

Table 2.2: Classification of the primary studies with respect to the Environments classes.

Study	Behavioural inputs			Behavioural outputs					Physical properties				Uncertainty		
	Requests								Spatial		Temporal				
	Declarative goals	Procedural goals	Preferences	Device operations	Human actions	Robot actions	Information services	Application services	Object locations	Human locations	Qualitative relations	Metric constraints	Unexpected changes	Action contingencies	Partial observability
S1		✘		✘											✘
S2	✘		✘	✘			✘	✘		✘				✘	
S3	✘		✘	✘	✘							✘	✘		
S4		✘		✘											
S5		✘	✘	✘					✘	✘					
S6	✘						✘			✘			✘	✘	
S7						✘									✘
S8		✘			✘							✘			
S9		✘	✘		✘					✘		✘	✘		
S10		✘	✘	✘											
S11	✘			✘											
S12			✘		✘				✘		✘				
S13		✘		✘											
S14	✘	✘	✘	✘					✘	✘	✘		✘	✘	
S15					✘				✘						
S16					✘					✘			✘		
S17		✘			✘							✘			
S18	✘			✘					✘	✘	✘			✘	✘
S19	✘	✘	✘	✘			✘	✘		✘					
S20		✘	✘		✘							✘	✘		
S21	✘			✘	✘		✘		✘	✘	✘				
S22	✘		✘	✘					✘	✘		✘	✘		
S23		✘		✘		✘				✘					
S24	✘			✘						✘					
S25	✘				✘			✘							
S26		✘		✘											
S17	✘			✘							✘				
S28	✘			✘		✘				✘	✘				
S29	✘				✘	✘			✘	✘		✘			✘
S30		✘					✘						✘	✘	
S31	✘				✘				✘	✘					
S32	✘			✘					✘	✘	✘				
S33		✘		✘					✘						
S34	✘			✘					✘				✘		
S35	✘			✘	✘		✘	✘							
S36		✘		✘					✘	✘					
S37				✘					✘						✘

achieve-maint($kitchenVentilator = ON \wedge$
$TvState = ALARM \wedge kitchenWindow = OPEN) \wedge$
achieve-final($doorsLeadTo(KITCHEN) = CLOSED$) under_cond_or_not
achieve-maint($personRoom \neq KITCHEN$)

Figure 2.2: Example of an extended goal (Kaldeli et al. 2012).

behavioural input, namely *Requests* and *Preferences*.

Requests

Request is the model of a desired result issued by a person or a software component for the purpose of achieving an explicit and specific behaviour, adaptation or organisation of a ubiquitous computing environment. The request therefore reflects directly the model of goals in a planning problem.

The notion of a declarative goal is well established in the planning community. Generally, a *declarative goal* specifies the state of an environment that needs to be established. Declarative goals are usually related to the question of what has to be achieved in a given setting. The majority of primary studies have a traditional and straightforward approach towards the understanding of a declarative goal, that is, a description of a final (goal) state (Ranganathan and Campbell 2004, Kotsovinos and Vukovic 2005, Pajares Ferrando and Onaindia 2013, Bidot et al. 2011, Song and Lee 2013, Krüger et al. 2011, Grześ et al. 2014, Heider 2003, Ortiz et al. 2013, Milani and Poggioni 2007, Garro et al. 2008, Marquardt and Uhrmacher 2009a). The remainder of classified studies incorporate extended forms of a declarative goal. Masellis et al. (2010) use so-called *planning programs* that include a finite set of maintenance and achievement goals expressed through propositional formulae. In (Kaldeli et al. 2012), an *extended goal* is a declarative and powerful expression on numerical variables, temporal constructs and maintainability properties. As an example, recall the gas leakage scenario in Theodore's home. An extended goal for such a situation is shown in Figure 2.2. It means that the kitchen ventilator must be turned on, the TV must show a warning and the window in the kitchen needs to be opened in some intermediate state and stay satisfied until the final state. The expressive power of the goal comes from its additional constructs. The proposition under `achieve-final` must be satisfied in the final state, but it may hold or not during the execution of the plan. On the other hand, the specification `under_cond_or_not` indicates that the doors should be closed only if the person is outside of the kitchen room, otherwise only the rest of conjunctions of sub-goals is considered.

Rocco et al. (2014) interpret a declarative goal through a constraint network that may include diverse types of constraints, such as temporal, resource, symbolical

and information dependencies. Further, declarative and constraint-based goals are proposed in (Cirillo et al. 2012). The goal, which is represented as logical formulae over the state, consists of several sub-goals. Each sub-goal is associated with a value denoting its importance of achievement. For example, if the goal of Tars is to clean the kitchen and bedroom, while more important is to clean the kitchen, it can be represented as $(dirt(kitchen) = 0, 0.6) \wedge (dirt(bedroom) = 0, 0.4)$. The flexibility of such a goal can be understood as the goal can be violated in some cases at the expense of a less efficient but valid solution. In addition to these reachability goals, maintenance goals are supported too (*cf.* interaction constraints). If maintenance goals are violated in all states, there cannot exist a valid solution. An example is to instruct Tars to never execute an action where Theodore is also present: `always` (`forall` r : (`not` (`exists` $h : robot - in = randhuman - in(h) = r)$)).

Fraile et al. (2013) provide an approach that is supposed to reach, maintain and carry out objectives. Reaching an objective denotes the conventional establishment of a final state. However, the maintenance of objective might be misleading as it refers to a function of the system and not to the power of expressing maintenance goals. Maintaining an objective means that a particular state will be re-established whenever it does not hold, which usually happens at the monitoring and execution phase, and not at planning time. Carrying out an objective refers to the planning process itself.

The second model of request is a procedural goal. A *procedural goal* basically specifies a set of procedures which are performed in order to satisfy an objective. Procedural goals are usually related to the question of how to accomplish something in a given setting. In contrast to the studies classified in the category of declarative goal, we observe that the studies in the current class adopt an exceptionally conventional approach towards using and implementing a procedural goal. The goal is either a single task (Ding et al. 2006, Sánchez-Garzón et al. 2012, Ha et al. 2005, Marquardt et al. 2008, Madkour et al. 2013, Jih, Hsu, Lee and Chen 2007), or a list of tasks (Qasem et al. 2004, Amigoni et al. 2005, Courtemanche et al. 2008, Bajo et al. 2009, Liang et al. 2010, Santofimia et al. 2010, Bidot et al. 2011, Hidalgo et al. 2011, Song and Lee 2013, Georgievski et al. 2013). In most of these studies, a task and a list of tasks are interpreted analogously to the definitions of a task and task network, respectively, within Hierarchical Task Network (HTN) planning (see Chapter 4). Amigoni et al. (2005) enhance a goal task with additional information on the performance, cost and probability of success for each decomposition of the task. The performance parameter expresses the expected effectiveness of a decomposition, the cost parameter indicates the amount of resource that would be consumed if the decomposition is applied, and the last parameter gives the expectation that no failures will occur when the decomposition is applied. While these parameters can

be indeed useful to ubiquitous computing environments, the drawbacks are that their semantics are not defined, and their values should be provided manually by a domain author.

Preferences

Preferences model individual attitudes or desires towards the behaviour or organisation of an environment. In contrast to a request, which must be achieved by the final solution, preferences are satisfied in the planning process as much as possible. A user may specify personal utilities for domain predicates (Ranganathan and Campbell 2004), select and customise a recipe (Kotsovinos and Vukovic 2005), choose different layouts on different presentation devices (Ding et al. 2006), specify preferred timetable (Bajo et al. 2009), indicate preferences on services (Liang et al. 2010), establish preferences on daily activities (Mastrogiovanni et al. 2010), encode preferences with respect to the current planning domain (Bidot et al. 2011), personalise services (Song and Lee 2013), define treatment preferences (Sánchez-Garzón et al. 2012), and indicate a personalised choice on various devices (Fraile et al. 2013). In (Ranganathan and Campbell 2004), each user preference is in a form of utility u for each predicate in different contexts, where $u \in [-10, 10]$. If a particular predicate does not have a preference value, then $u = 0$. Therefore, each user has a utility for each state of the environment, whereas the utility of the goal state is a linear combination of the utilities of all entities relevant to the goal. The utility of the environment is defined as a linear combination of individual utilities of the states of all entities present in the environment. However, the incorporation, maintenance and handling of such preferences becomes cumbersome when the number of predicates and the number of users increase.

Behavioural outputs

We use the term *behavioural output* to refer to the particular way in which the information that changes the state of an environment is represented and produced. Practically, the behavioural output is used to encode the domain knowledge for a planning system. We identify five classes of behavioural outputs, namely *Device operations*, *Human actions*, *Robot actions*, *Application services*, and *Information services*. The classes, which are mutually exclusive, are defined as follows.

Device operations

We define a *device operation* as a functionality that a specific device can perform. By device we mean a piece of equipment deployed in ubiquitous computing environments that has limited capabilities to interact autonomously with (other devices

```
(:action turn-on-device
 :parameters (?d - device)
 :preconditions (and (not (turned-on ?d)) (other_cond))
 :effects (turned-on ?d))
```

Figure 2.3: Template action for turning on a device represented in PDDL.

in) the environment. Examples of devices include a laptop, projector, smoke alarm, gas extractor, air conditioner, canvas, automatic blind, TV set, lights, screen, heating system, computer, printer, *etc.*

We identify two groups of primary studies with respect to their conception of a device operation. In the first group, a device provides its operation in the form of an action with preconditions and effects (Ranganathan and Campbell 2004, Amigoni et al. 2005, Ding et al. 2006, Fraile et al. 2013, Krüger et al. 2011, Heider 2003, Rocco et al. 2014, Milani and Poggioni 2007, Georgievski et al. 2013, Garro et al. 2008, Harrington and Cahill 2011). As an example, in Figure 2.3, we provide a template action for turning on a device encoded in PDDL. Preconditions usually represent the state in which the device must be so as to achieve the desired behaviour. In addition to the device state, preconditions may encode other environment properties. For example, a device's spatial attribute, which includes the device location and the region of the environment over which the action has effects, can be used to annotate the preconditions with additional semantics (Harrington and Cahill 2011). Such preconditions must hold in the current state of the environment in order that the action can execute its effects and modify the environment in a future state. A template action for turning off a device can be modelled similarly.

Ding et al. (2006) find it necessary to describe each device with a pair of action and schemata. The action corresponds to a planning action, while schemata contains a set of instructions that are sent to the respective devices after the planning process has finished and under the assumption that the action is part of the solution. In addition to planning actions, Milani and Poggioni (2007) make use of a reactive operator. A reactive operator has a set of triggering conditions that activate the operator and a set of effects that are applied as a result of the reactive behaviour. Reactive operators are used to represent reactive devices, that is, devices with "stimulus response behaviour".

In the second group, the notion of service is used to represent and execute device operations, while a single device may offer one or more services (Qasem et al. 2004, Kotsovinos and Vukovic 2005, Liang et al. 2010, Masellis et al. 2010, Santofimia et al. 2010, Bidot et al. 2011, Kaldeli et al. 2012, Ha et al. 2005, Marquardt et al. 2008, Marquardt and Uhrmacher 2009a, Jih, Hsu, Lee and Chen 2007). Santofimia et al. (2010) propose a semantic model for the relationships between devices, actions and

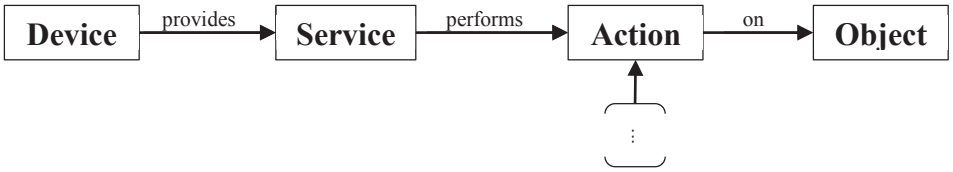


Figure 2.4: The relationship between devices, actions and services (Santofimia et al. 2010).

services.[2] Figure 2.4 illustrates the model and relationships between respective entities. From our point of interest, a device provides a service that performs an action on one or more objects. An object is an artefact found in a ubiquitous computing environment.

Kaldeli et al. (2012) require each device to expose its functionalities as one or more Universal Plug and Play (UPnP)[3] services. UPnP services provide a set of method calls that constitute the set of UPnP actions, where a UPnP action can have several input and output parameters. For the purpose of planning, UPnP services are translated into planning actions and augmented with additional semantics in the preconditions and effects. Marquardt and Uhrmacher (2009a) encode a service as a PDDL action directly.

Human actions

We define a *human action* as a behaviour to be achieved by a person within an environment. Human actions are used to assist or guide people on the path to accomplish their goals or activities. Human actions may be part of the domain model, and therefore, they can be used by planning systems to create human-aware plans. Plan actions then serve as notifications or help advice to individuals accomplishing their tasks. Human actions are used in the domain of assisted cooking (Kotsovinos and Vukovic 2005, Sando and Hishiyama 2011, Ortiz et al. 2013), and assisted daily living (or assisted care giving) (Mastrogiovanni et al. 2010, Yordanova 2011, Cirillo et al. 2012, Courtemanche et al. 2008, Bajo et al. 2009, Hidalgo et al. 2011, Sánchez-Garzón et al. 2012, Pajares Ferrando and Onaindia 2013, Grześ et al. 2014). While in Figure 2.5 we show a template human action for picking up an object contained in some place as provided in (Ortiz et al. 2013), Yordanova (2011) extracts and generalise sixteen template human actions from a set of domains. One specific representation of human actions is through the use of affordances and capabilities, which are regions in a proper space characterised with a set of attributes (Mastrogiovanni et al. 2010). For example, an object affording a capability "to take" is a region

[2] An extended and formalised version of the model can be found in (Santofimia et al. 2011).
[3] `www.upnp.org`

```
(:action pick-up
 :parameters (?param1 - moveable ?param2 - surface)
 :precondition (and (in ?param2 ?param1))
 :effect (and (not (in ?param2 ?param1)) (holding ?param1)))
```

Figure 2.5: Template action for picking up an object by a human encoded in PDDL (Ortiz et al. 2013).

in the respective affordance space characterised by the weight and grasp size attributes. In the planning context, people have initial capabilities and can acquire new capabilities by using object affordances. Say that the object is a vacuum cleaner and Theodore has the initial capability to take it. By taking the vacuum cleaner, Theodore acquires new capabilities such as "to clean".

Given a state s of a ubiquitous computing environment, a *human action* is a tuple $\langle pre, t, \phi \rangle$, where $pre \subseteq S$ are preconditions, $t \in \mathbb{R}^+$ is action duration, and $\phi : S \times S \rightarrow [0, 1]$ is a transition relation such that $\phi(s, s')$ indicates the probability of transition from state s to state s' by performing the action. In (Cirillo et al. 2012), this definition is successful under the assumption that the duration of an action is fixed, and the action cannot be interrupted once started. Preconditions are not required, but if they are considered, then they are verified only at fixed points in time. Therefore, human actions with preconditions are always instantaneous.

Robot actions

We define a *robot action* as a behaviour of a robot performed within a ubiquitous computing environment intended to achieve some goal. Robot actions can be categorised in two groups, namely, actions that transform the environment and actions that are communicated to people. Since basically robots are devices, their actions can be considered as device operations. In contrast to what we consider a device, a robot is autonomous and intelligent to a certain degree, hence robot actions represent a separate class.

Carolis and Cozzolongo (2007) describe a robot action by sets of preconditions and possibly non-deterministic effects. Both sets are associated with probability values. The value of preconditions indicates the probability that preconditions will hold in the current state, while the value of effects demonstrates the probability that the action will have the effect in the current state. In addition, effects may include values about expected social utility and expected task utility of an action. Rocco et al. (2014) represent robot actions (*cf.* functionalities) as state variables whose instantiation either produces information or modifies the environment. A robot functionality may require some information input and may consume resources. The interactions and dependencies of functionalities with respect to information and

```
name: robot-clean(r)
precond: room(r) and dirt(r) > 0
results: dirt(r):=(dirt(r) - 1) and cost:=2
time: 10
```

Figure 2.6: Example of a robot action (Cirillo et al. 2012).

```
(:action set-file-ppt
 :parameters (?id - string ?file - pptfile)
 :vars (?mac - machine ?slide - number)
 :precondition (and (ppt ?id ?mac started ?file1 ?slide)
                    (not (= ?file ?file1)))
 :effect (ppt ?id ?mac started ?file 1)
 :check ((= (get-file-ppt ?id) ?file) (failure non-retryable)))
```

Figure 2.7: Example of an application service (Ranganathan and Campbell 2004).

resource requirements are represented with temporal relations using Allen's interval algebra (Allen 1983). Further, similarly to human actions, Cirillo et al. (2012) encode a robot action through the use of preconditions, effects, cost and a fixed duration. A robot may move to a specific location, sleep for a certain period, stay in some position, clean a room, and remind a person to take medication. Figure 2.6 demonstrates the action a robot should take in order to clean a room specified at planning time. Finally, Ha et al. (2005) represents robot actions as (Web) services, that is, atomic processes in the Semantic Markup for Web Services (OWL-S) terminology (Martin et al. 2007). At the planning level however, a robot action (an atomic process) is encoded as a primitive task in the scope of HTN planning.

Application services

We define an *application service* as a purposeful behaviour of a piece of software, that is, an application, installed on a machine deployed in a ubiquitous computing environment. Applications might be commercial, such as Microsoft Power Point, Adobe Acrobat and Apple Keynote, or developed specifically for the needs of the respective environment.

Four primary studies fall in the class of application services. Ranganathan and Campbell (2004) define an application service as an invocation of a method on an application. The application service is represented as a PDDL action with preconditions and effects. Figure 2.7 shows a service that sets a file on a particular machine for the purpose of presentation by using the Microsoft Power Point application. Preconditions state that the application should be started on some machine and the file displayed on that machine should be different from the one in the input parameter. The effect ensures that the current file is started on slide number 1.

```
(def-adl-operator (make_restaurant_booking ?r ?ppl ?t)
 (pre (restaurant ?r) (rest_found ?r)(rest_booking_online ?r ?e)
      (rest_has_space ?r ?ppl)(persons ?ppl) (time ?t)
 (and (not (rest_booking_made ?ppl ?t))(not (rest_booked ?r))
      (restaurant ?r) (persons ?ppl)(time ?t)))
 (add (rest_booking_made ?ppl ?t)(rest_booked ?r)))
```

Figure 2.8: Example of an information service encoded in Action Description Language (ADL) (Vukovic et al. 2007).

Information services

We define an *information service* as a knowledgeable behaviour built by collecting, managing and reasoning over possibly distributed data, and showed to users. Although not necessarily, such information service is usually represented as a Web service. The Web service is then translated into a planning action suitable for the respective planning system.

Ranganathan and Campbell (2004) identify a service as a way of acting in an environment, however, the notion of service is neither defined nor exemplified. Vukovic et al. (2007) do not define the notion of a service, but they make use of Web services to provide information delivered to mobile phones. In the scenario described, information services include things like a restaurant finder, which provides a list of available restaurants, a direction finder, which navigates a user to a given restaurant, a translator service, which translates a specific content from one language to another, and a speech-synthesizer service, which converts text to speech. The planning action shown in Figure 2.8 encodes an information service that can be used by Samantha to book a restaurant for Theodore and his friends in a particular time slot of the day.

Song and Lee (2013) provide a rather general description of what a service does, that is, a "service performs tasks using its own functions", where examples of tasks are converting currency and getting weather information. It also remains undefined how services are represented at the planning level. Madkour et al. (2013) describe a service as a function provided by the middleware and invoked by a mobile application. A service is associated with several policies each of which represents a method used to deliver a service under specific resource requirements and quality-of-service conditions. Marquardt and Uhrmacher (2009a) incorporate information services represented by Web services. At the planning level, information services are encoded as PDDL operators. Jih, Chen and Hsu (2007) employ information services in form of Web services. An example provided is a service that access Google Calendar and provides information about the schedule of a person. Generally, a service functionality is semantically annotated with its purpose and functionally

described with its way of operating. A service (profile) is translated into an operator associated with preconditions and effects. Finally, Pajares Ferrando and Onaindia (2013) use information services through the functionality of Google Maps.

Physical properties

We define *physical properties* as the information used to characterise a situation of a person, object or a place with respect to space and time. Practically, physical properties are subsumed under the context information of a ubiquitous computing environment where people, objects and places are the entities of the environment (Abowd et al. 1999). Context information, which is encoded within domain and problem specifications, is used by planning systems to search for solutions that support context-aware behaviour of the environment. The respective subclasses of the Physical properties class are *Spatial properties* and *Temporal properties*.

Spatial properties

The term ubiquitous in ubiquitous computing refers to the computation being available everywhere. People, sensors, and actuators have a physical extension that relates them to one another and with space. A *spatial property* qualifies the relation among components of the system and their environment. Most planning systems include a – more or less elaborate – form of spatial representation which defines the types and qualities of the spatial properties, in turn, allowing for proper reasoning over these (Aiello et al. 2007a). There are two major categories of spatial representations, those that respect the spatial characteristics of the underlying spatial models and those which treat space simply as a set of symbols without considering any geometrical or physical laws (Aiello et al. 2007b). We call the first type *purely spatial representation* while the second ones are *abstract spatial representations*. For instance, consider a sphere containing n actuators $\alpha_1, \ldots \alpha_n$ of the same radius so that they all touch the same spheric unit actuator α_0. In an abstract representation, where the actuators are represented as a set of points, the number n can be arbitrary – something that is not physically realisable (Pfender and Ziegler 2004). On the other hand, if space is properly represented, one would require that actuators do not overlap and at most 12 can be arranged in such a physical configuration. The issue becomes yet more interesting when time and space are represented together in the planning system.

Consider Theodore moving from his bedroom to the kitchen. In a purely abstract representation, one can represent the move as instantaneous. In a 'pure' representation, this would not be possible and the fact that the locations have to be topologically connected and that nothing can travel at infinite speed would have

to be taken into account. There is thus an issue related to spatial arrangements in plans elsewhere known as spatial *realisability* (Lemon and Pratt 1997, Kontchakov et al. 2014), and at times also the need for the consideration of spatio-temporal properties (Andréka et al. 2007).

In performing the analysis, we notice that the abstract approach is often the one taken. Typically, some form of structure is given to the spatial concepts, such as a hierarchy of locations that represents the being-part-of relation (mereology (Whitehead 2010)), and, at times, also includes connectedness information (mereotopology (Casati and Varzi 1999)). Though, these models are weak from the realizability point of view, they can already offer sufficient domain knowledge for proper planning in ubiquitous computing.

The class Spatial properties is divided in two subclasses. *Object locations* class comprises primary studies that consider locations of objects, such as devices and sensors, for the purpose of planning. In most cases, the locations of objects represent static information predefined in the calibration phase of the environment. Although all these studies claim to take spatial properties of objects into account, it is extremely difficult to interpret the way in which these properties are represented at the planning level. Several studies use abstract (relative) models to describe spatial properties of objects (Kaldeli et al. 2012, Pajares Ferrando and Onaindia 2013, Ortiz et al. 2013, Milani and Poggioni 2007, Georgievski et al. 2013, Jih, Hsu, Lee and Chen 2007). A general form of a spatial property for objects is a simple predicate `op(p1,...,pN)`, where `op` is a spatial operator and `p1,...,pN` are parameters, both objects and locations. As for the operator, it can take one of the values: `at` (Pajares Ferrando and Onaindia 2013, Milani and Poggioni 2007), `pos` (Pajares Ferrando and Onaindia 2013), `in` (Ortiz et al. 2013, Georgievski et al. 2013, Jih, Hsu, Lee and Chen 2007), `distance`, `near-by`, *etc.* An example is the predicate `in(TV,living-room)` meaning that the TV is in the living room. Another example is a typified predicate `pos(?a - ambulance) - address` that represents the position of an ambulance as an address. These spatial-related predicates are usually used to model the domain knowledge, including preconditions of actions (*e.g.,* to check whether a necessary object is present in the right location), effects of actions (*e.g.,* if the action enables displacement of an appropriate object), and other domain-specific knowledge. In many cases, it is not the only requirement that objects, or more specifically, devices are at a specific location, but also the knowledge about how they affect the objects at or near by the device location (Milani and Poggioni 2007, Harrington and Cahill 2011). An example of such a device is a lamp.

Mastrogiovanni et al. (2010) use capabilities and affordances to represent the fact that an object is in a particular location. A representation scale is used in which level one corresponds to furniture and containers inside rooms, while different rooms

represent the second level of the scale, which corresponds to the building interior. Finally, Harrington and Cahill (2011) use a geometric model to represent spatial relationships of sensor and actuators.

Human location defines the position of a person within a ubiquitous computing environment. Human location, as dynamic information, is an important factor in many situations: to adjust the environment as people move around, to maximise safety of people, to generate better medical plans, (robots) to respect the presence of people in particular places, *etc.* The location of people can be tracked to the level of some predefined areas, such as rooms. More fine-grained information about the human location can include the posture and orientation of the human. The human location specifying the human posture possibly in some orientation is important for many scenarios such as those concerning the safety of people (*e.g.,* fall detection and appropriate reaction). However, more fine-grained spatial information may become a subject to privacy concerns, which have to be taken into account when gathering and reasoning over that information (Bettini and Riboni 2015).

The primary studies take the former approach. A general form of a simple predicate denoting a location occupied by a person is `human-in(human,location)`. Similarly to object locations, a human location may be given in the form of an address, say, in a predicate whose type is address. A human location can be extracted from Radio Frequency Identification (RFID) tags attached to people (Bajo et al. 2009, Fraile et al. 2013, Ha et al. 2005, Ortiz et al. 2013), people's mobile phone locations (Song and Lee 2013), Global Positioning System (GPS) (Sando and Hishiyama 2011, Fraile et al. 2013), Ubisense system (Krüger et al. 2011), and other location tracking systems (Kaldeli et al. 2012, Rocco et al. 2014, Jih, Hsu, Lee and Chen 2007).

Temporal properties

Time is an important dimension in ubiquitous computing environments given that the duration of people's activities can be personalised, the environment may take shifts of staff and their scheduled activities into account, better response time of the environment can be achieved due to potential of high parallelism of independent actions, and so forth. Here, a *temporal property* refers to the way of representing and organising activities with respect to time (Benthem 1983). At the planning level, one way to model temporal properties in ubiquitous computing environments is to use the notion of durative actions. The primary studies that employ this notion are classified under the *Metric constraints* subclass. Generally, temporal properties can be expressed through annotations of all preconditions and effects in durative actions. Although it is difficult to extract meaningful information from the primary studies, the temporal annotation of preconditions indicates explicitly when the associated fact must hold: at the *start* of the interval (*e.g.,* starting time of

tasks (Bajo et al. 2009), start interval for tasks (Sánchez-Garzón et al. 2012), and early start time of activities (Fraile et al. 2013)), at the *end* of the interval (*e.g.,* ending time of tasks (Bajo et al. 2009), end interval for tasks (Sánchez-Garzón et al. 2012), late end time (Fraile et al. 2013)), or over the interval. The temporal annotation of effects signifies that the effect is immediate, when it happens at the start of the interval, or delayed, when it happens at the end of the interval. Thus, two actions are simultaneous if they are completely or partially executed within the same interval of time. A simplified case is when only the duration of an action is modelled and taken into account during the planning process. An example of such durative action is shown in Figure 2.6.

Another way, which is represented by the *Qualitative relations* subclass, is to use qualitative constraints to express temporal relationships between various activities. Rocco et al. (2014) model qualitative constraints by using the relations of Allen's interval algebra, such as *before*, *meets*, *overlaps*, *etc.* An example relation is $a_2\ meets\ a_1$ which represents that actions a_2 ends as soon as action a_1 starts.

Temporal properties modelled in planning domains provide a possibility for planning techniques to create plans with concurrent actions naturally. Even if such explicit encoding of temporal properties is not provided in a domain model, an automated planner may induce that the order of actions in the final plan can be partial. The latter case, which can be identified in several primary studies (Mastrogiovanni et al. 2010, Bidot et al. 2011, Kaldeli et al. 2012, Pajares Ferrando and Onaindia 2013, Heider 2003, Milani and Poggioni 2007), is also classified within the class of Qualitative relations. A partially ordered plan could be defined as a tuple $\langle A, \prec, CL, SC \rangle$, where A is a set of actions, $\prec$ is a set of ordering constraints between actions in A, CL is a set of casual links between actions in A, and SC is a set of supportive constraints, such as variable constraints (Bidot et al. 2011) or support links (Pajares Ferrando and Onaindia 2013).

Uncertainty

Uncertainty is a broad concept, but it can be perceived as a three-dimensional property of ubiquitous computing environments: a dimension of capacity, complexity and dynamism. The capacity defines the potential of environments to constantly expand, for example, new devices are deployed. The complexity refers to the heterogeneity of entities (for example, devices) present and activities happening in the environment, while the dynamism comprises predicted and unexpected events, changes or failures of sensors and devices, and people behaviour over time. We focus here on the issues closely related to the third dimension and we divide the Uncertainty class in three subclasses: *Unexpected events*, *Action contingencies*, and *Partial observability*.

Unexpected events

The context of ubiquitous computing environments is characterised by diverse and continuous events. An event that happens in an exceptional situation within an environment is considered as an *unexpected event*. For example, unavailability of resources, changes in patient condition, a person choking, a patient's unexpected visit, or a user lowering the radio volume just before the system is supposed to do it. At the planning level, unexpected events may satisfy effects of already planned (scheduled) actions (Vukovic et al. 2007, Madkour et al. 2013), or invalidate preconditions that were true during planning or action instantiation (Vukovic et al. 2007, Bidot et al. 2011). Such unexpected events basically interrupt the execution of a plan. For example, if a necessary ingredient is no longer available in Theodore's home, the plan, which is in form of a recipe, cannot be further executed, and requires appropriate adaptations (Kotsovinos and Vukovic 2005). An adaptation may be achieved by using manually encoded conditional statements (Sando and Hishiyama 2011), a repair of the current plan given some predefined knowledge (Bajo et al. 2009, Sánchez-Garzón et al. 2012, Fraile et al. 2013), and replanning (Kotsovinos and Vukovic 2005, Vukovic et al. 2007, Bidot et al. 2011, Sánchez-Garzón et al. 2012, Fraile et al. 2013, Madkour et al. 2013).

Action contingencies

It is not a rare situation when the execution of plans does not proceed as expected (during the planning process) due to the non-determinism concerning the actual output of plan actions. We define an *action contingency* as the state of an action in which the action does not work correctly during execution.

There are two types of action contingencies, namely *action failure* and *action timeout*. The former one happens when an action invocation returns an erroneous response (Ranganathan and Campbell 2004, Vukovic et al. 2007, Bidot et al. 2011, Kaldeli et al. 2012, Madkour et al. 2013). In practice, an erroneous response may indicate a simple failure, permanent failure, or Byzantine behaviour – an action completes successfully without providing the expected result. Timeout occurs when an action invocation does not provide a response after a certain amount of time within which, for instance, an average fast action is expected to respond (Vukovic et al. 2007, Bidot et al. 2011, Kaldeli et al. 2012, Madkour et al. 2013). Most primary studies identify that action timeouts are due to the disconnection of the network which devices providing the actions are part of. A timeout is interpreted as an erroneous behaviour, and it is therefore handled the same way as action failures.

Action failures and action timeouts can be handled depending on the type of faulty action (Ranganathan and Campbell 2004, Vukovic et al. 2007), availability

of alternative action instantiations (Vukovic et al. 2007, Bidot et al. 2011, Madkour et al. 2013), and the severity of reported failure (Kaldeli et al. 2012). Ranganathan and Campbell (2004) use additional information encoded in specific actions. One piece of additional information is the return value of an action invocation to indicate whether the action has failed. Another piece of information indicates whether the planning system can try to execute the action again, but only if it has failed once before. The most common approach to handle simple action failures is to retry the execution of the same action a specific number of times, and in case of reaching that limit, to initiate replanning. Section 2.3.2 provides more details on the approaches taken to handle uncertainty. Finally, timeout conditions may differ depending on the type of actions (Kaldeli et al. 2012). For example, an action invoked to close a door may respond in orders of seconds faster than an action to pull down window blinds.

Partial observability

Partial observability refers to the imperfectness and incompleteness of information that a planning system has about the state of a ubiquitous computing environment. This means that the planner cannot make the closed-world assumptions (Ghallab et al. 2004). First, state constituents, say, variables may have different possible actual values, and second, planning operators can no longer depend directly on such uncertain state. We identify two major approaches to solve these issues. One approach is to use sensing so as to gather the actual values of unknown variables, while the other one is to express values as probabilities rather than certain observations. The former approach implies a use of sensing actions (also information-gathering actions or observational actions), while the latter one relies on belief states.

Qasem et al. (2004) gather information based on local completeness of information and the relevance of information sources. Completeness is represented through description logic expressions over classes of objects with similar characteristics. For some device associated with knowledge base belonging to some class, the completeness indicates that if some entity is included in the knowledge base, it is not an instance of the respective class, but it is in the complement of that class. When there is insufficient information to validate conditions, information must be sensed from relevant information sources. Kaldeli et al. (2012) represent a smart home through a dynamic constraint network which provides naturally the possibility to update the current state of the home (without reloading the planning domain). Since the planner is informed about the changes of the home state, it searches for a plan given the initial state for which it has complete knowledge. Two main assumptions are made, namely (1) continual sensing is performed – sensors check and publish their

status in regular time intervals; and (2) information is persistent – sensed information involved in actions to be executed is valid until the execution finishes.

Several primary studies deal with partial observability using belief states based on observations (Carolis and Cozzolongo 2007, Cirillo et al. 2012, Harrington and Cahill 2011). An observation o can be perceived as a set of literals, while the probability of having o when some action a is performed and the resulting state is s is given by $f_a : S \times O \rightarrow [0, 1]$ such that $o \in O$ and O is a set of observations (Cirillo et al. 2012, Harrington and Cahill 2011). For instance, Cirillo et al. (2012) define a belief situation as a probability distribution over a number of situations, and observations as results of actions. If a is a robot action, then o is considered as a sensing action (*e.g.*, check the status of a door). If a is a human action, then o is considered as an indirect observation of a by a robot (*e.g.*, if the radio is turned off, Tars can observe that the `listenToRadio` action has finished).

2.3.2 Planning

The class of *Planning* encompasses the core concepts and aspects of AI planning. It deals with the purpose of using planning, the types of planning techniques, the definition and representation of planning problems, modelling aspects, design and implementation of planning systems, integration of such systems into ubiquitous computing architectures, and the use of planning systems at a particular stage of the life cycle of ubiquitous computing systems. Each such concept and aspect is represented by its own class. The classification of the primary studies with respect to the Purpose class and the Planning technique class can be found in (Georgievski and Aiello 2015b), while the classification of the studies with respect to the rest of classes is shown in Table 2.3. We identify that about 60% of the studies mention the planning problem being addressed, and that only half of the studies provide information about the problem representation. Only four studies focus on domain modelling, and half of the studies describe the actual problem definition. Moreover, only six studies pay attention to expressiveness constructs potentially useful for ubiquitous computing, and about 60% of the primary studies reveal the language used to model the domain, state, and goal. Furthermore, about 45% of the studies deal with monitoring of the plan execution and recovery in the case of contingencies. As shown in Table 2.3, a small number of the primary studies fall into the classes of Life cycle and Integration architecture, but, on the other hand, about 70% of the studies are classified within the Design and implementation class.

Table 2.3: Classification of the primary studies with respect to the Planning classes.

		Prob. rep.		Modelling							Life cycle			Development			
							Languages								Integ. arch.		
Study	Planning problems	Classical	State-variable	Domain models	Problem definitions	Expressiveness constructs	Knowledge	Goals	States	Monitoring and recovery	Design-time planning	Compile-time planning	Runtime planning	Design and implementation	Multi-agent	Modular	Service-oriented
S1	✘	✘								✘				✘			
S2		✘			✘		✘	✘	✘	✘				✘		✘	
S3	✘	✘			✘	✘	✘	✘	✘	✘			✘	✘			
S4	✘				✘									✘	✘		
S5	✘				✘		✘			✘							
S6		✘			✘	✘	✘	✘	✘	✘	✘			✘			
S7					✘					✘							
S8	✘															✘	
S9	✘									✘			✘	✘	✘		
S10														✘			
S11	✘	✘															
S12	✘													✘			
S13	✘						✘							✘	✘		
S14	✘	✘			✘		✘		✘	✘	✘			✘			✘
S15						✘	✘										
S16										✘							
S17	✘			✘	✘				✘					✘			
S18	✘		✘		✘	✘	✘	✘	✘	✘			✘	✘			✘
S19	✘						✘	✘	✘					✘			
S20							✘	✘		✘				✘		✘	
S21	✘		✘				✘	✘	✘						✘		
S22					✘					✘			✘	✘	✘		
S23					✘		✘		✘					✘			
S24		✘					✘					✘					
S25	✘		✘	✘	✘									✘			
S26													✘				
S27	✘	✘				✘	✘	✘	✘					✘			
S28			✘		✘					✘				✘			
S29	✘		✘							✘				✘			
S30	✘				✘					✘	✘			✘			
S31	✘	✘		✘	✘		✘										
S32	✘	✘	✘				✘	✘	✘					✘			
S33	✘	✘		✘	✘		✘	✘	✘				✘	✘			✘
S34										✘						✘	
S35	✘	✘				✘	✘	✘	✘				✘	✘			
S36					✘		✘	✘	✘					✘	✘		
S37	✘		✘		✘		✘		✘								

Purposes

The *Purposes* class defines what planning is used for in an environment specific to the study under examination. We identify three types of purpose, namely *control*, *assistance*, and *organisation*.The *Control* class specifies that AI planning is used to create a sequence of actions whose execution does not involve human intervention. More than half of the primary studies employ planning to produce a control sequence. Several of these studies include control of robots (Carolis and Cozzolongo 2007, Cirillo et al. 2012), or an interoperation between robot and device actions (Ha et al. 2005, Rocco et al. 2014), while the rest of the studies deal with coordination of device actions.

In contrast to the Control class, the *Assistance* class indicates that planning produces a sequence of actions that either is aware of people's activities, or provides help and guidance to people during the process of goal accomplishment. This class considers direct human interaction. About one third of the primary studies adopt planning in order to create solutions that are human centred. One study creates a so-called human-aware plan, which does not include human actions, but only robot actions. However, the human-aware plan is generated based on a forecast of human actions and constraints on human interaction. The rest of classified primary studies include human actions in the solution plan that enables to alert people (Kotsovinos and Vukovic 2005, Sando and Hishiyama 2011, Grześ et al. 2014, Marquardt and Uhrmacher 2009a), or to guide them (Vukovic et al. 2007, Courtemanche et al. 2008, Mastrogiovanni et al. 2010, Yordanova 2011, Hidalgo et al. 2011, Sánchez-Garzón et al. 2012, Pajares Ferrando and Onaindia 2013, Fraile et al. 2013, Ortiz et al. 2013). These studies create a solution plan as a list of daily activities that elderly people should follow (Courtemanche et al. 2008, Mastrogiovanni et al. 2010), or as a care plan for patients that is performed by the patients themselves (Sánchez-Garzón et al. 2012, Fraile et al. 2013, Grześ et al. 2014), caregivers (Yordanova 2011), or both (Hidalgo et al. 2011, Pajares Ferrando and Onaindia 2013). Marquardt and Uhrmacher (2009a) advocate imperceptibly the exclusivity of plans either only to actions directly executable by the planner or only to human actions, which are expected to be executed by humans. Considering that authors use planning to compose services, then modelling and implementing human actions as services becomes a problem that cannot be solved under the umbrella of service composition.

The *Organisation* class indicates that planning outputs a sequence of tasks used to manage specific resources within a ubiquitous computing environment. Bajo et al. (2009) use plans to improve the management in a hospital. The plans are used to organise dynamically nurses' working time, to manage standard working reports about the activities of nurses, and to guarantee that patients assigned to particular nurses receive proper care. Similarly, Yordanova (2011) uses an organisational plan

to arrange the activities of a nurse for the purpose of taking care of a patient. The scenario illustrated in the study envisions also other support for the care personnel, such as automated management of documentation. Besides the plan with activities for a patient, Hidalgo et al. (2011) provide an organisational plan for a care centre based on the current context, available resources (for instance, rooms and timetables) and staff, and the organisation rules of the centre. An organisational plan is communicated to caregivers, such as nurses.

Planning techniques

The class of *Planning techniques* analyses techniques used to realise planning. A straightforward observation is that HTN planning is frequently adopted. Ding et al. (2006) find HTN planning "suitable for writing and solving presentation planning problems" because a task corresponds directly to creating a presentation, while Amigoni et al. (2005) use HTN planning as it is "considered as the most suitable for real-world applications", including ubiquitous computing applications. Several other primary studies offer more fair and elaborated reasons for the suitability of hierarchical planning, such as causality (Yordanova 2011), flexibility (Qasem et al. 2004), and effectiveness (Marquardt et al. 2008). On the other hand, Kaldeli et al. (2012) criticise HTN planning due to the requirement for a set of predefined methods that cannot be easily reconfigured when changes in the context, domain, or user requirements occur. Similarly, Marquardt et al. (2008) recognise a critical point in the use of hierarchical planning due to its need for methods. The problem with this necessity comes to light in real-life applications of hierarchical planning, when someone needs to be responsible for providing reasonable knowledge.

Probabilistic planning uses probabilities associated to non-deterministic events to search for a plan. Since actions are non-deterministic, probabilities are used to quantify the costs and successes of plans. A plan would be efficient if its cost does not surpass the benefit of reaching a goal. In cases when there is no plan to reach a goal, action probabilities are particularly useful in order to maximise the probability of reaching the goal. While probability planning is computationally complex as probabilities create a belief state that is continuous and infinite, there are still several primary studies that take advantage of it. Carolis and Cozzolongo (2007) use a simplified probabilistic planning and associate probabilities to goals of a robot. Their probabilistic model is based on Bayesian Networks (BNs) and captures the uncertainty and partial observability of a ubiquitous computing environment. The plan selected to achieve the goals maximises the expected utility giving the probability outcomes of the variables as computed in the goal BN. Courtemanche et al. (2008) use a Markov decision process (MDP) (Puterman 1994) to represent a probabilistic planning problem. Given such planning problem, the planner searches for

a plan that maximises the expected reward accumulated over some horizon of interest. Grześ et al. (2014) employ probabilistic planning by using a temporal probability model based on partially observable MDPs (POMDPs). Their POMDP model is based on environmental observations in order to deal with the uncertainty coming from unpredictability of human behaviour, and noise and inconsistency of sensor readings. Cirillo et al. (2012) base their approach on probabilistic planning that generates plans conditional on observations related to the actions of humans within a ubiquitous computing environment. This probabilistic and partially observable planning approach is an extension of the PTLplan planner (Karlsson 2001) to reason over belief situations and reach a situation in which a plan with satisfactory value is found. Finally, Harrington and Cahill (2011) envision probabilistic planning to be used in their models. The objects modelled in actions are associated with a probability value that indicates the confidence of a successful state transition of an action.

CSP-based planning assumes that a planning problem can be encoded as a constraint network which in turn represents a Constraint Satisfaction Problem (CSP) whose inconsistencies are to be solved by a constraint solver. A CSP consists of three finite sets, specifically, V is a set of variables, D is a set of domains of the variables in V such that $v \in D^v$, and C is a set of constraints over the variables in V. A constraint involving variables from V represents a restriction over the values that can be assigned to those variables. The solution to a CSP is a valuation of each variable $v \in V$ with a value from D^v such that all constraints in C are satisfied. Kaldeli et al. (2012) adopt the approach we just described. Rocco et al. (2014) represent a goal as a constraint network that needs to be refined into a consistent and feasible one. To that end, a so-called meta-CSP approach is adopted: the problem of refining the goal constraint network is cast as a high-level CSP, which builds on lower-level CSPs (each for a particular feature, such as temporal, causal, resource inconsistencies). CSP-based planning can be particularly useful for ubiquitous computing due for the following peculiarities (Kaldeli et al. 2012, Rocco et al. 2014).

- *Numerical variables* – CSP-based planners are able to handle variables that range over large domains efficiently. Such variables are common in ubiquitous computing environments. For instance, variables for temperature measurements, TV channels, locations, *etc.*

- *On-line sensing* – since a constraint network naturally supports adding and removing constraints on the fly, updates of the current state of an environment can be performed efficiently. The updates, that is, sensed information, can have varying levels of abstraction: low-level observations, such as `on(stove)`, filtered state estimates, such as `at(user, location)`, or high-level interpretations, such as `cooking(user)`.

- *Continual planning* – a constraint network that supports adding and removing constraints dynamically fosters the interleaving of planning and execution.

- *Various interrelationships* – CSP-based planners support modelling of casual, temporal, resource and information dependencies between objects and actions taking part of an environment. For instance, actions may be subject to deadlines, or they may include spatial information and resources, which may be crucial to reasoning.

Partial-order planning in combination with defeasible argumentation is proposed in (Pajares Ferrando and Onaindia 2013). Partial-order planning, which is based on the least-commitment strategy (Weld 1994), postpones commitments of ordering among plan actions until these commitments become necessary. A plan therefore represents a set of actions and a set of constraints defining the order among actions. Two reasons can be identified to prefer partial-order planning over other planning techniques, namely (1) partial-order planning offers "a more promising approach" to deal with durative actions, temporal and resource constraints (Pajares Ferrando and Onaindia 2013, Smith et al. 2000); and (2) it offers a high degree of execution flexibility due to its support for parallelism.

Case-based planning is considered as planning supported by a changing dynamic memory (Hammond 1989). A case is a past experience consisting of an initial problem, a sequence of actions that solves the problem, and the final state achieved after the solution is applied. A case-based planner creates and modifies plans based on planner's past experiences which represent memory of effects (rather than causal rules). For instance, Bajo et al. (2009) consider tasks, resources and time as memories. Generally, memories of past successes are used by the planner when creating new plans, memories of past failures are used to inform the planner about potential problems, and memories of past repairs are used to instruct the planner how to handle repairs. Given a planning problem, case-based planning searches for a solution by taking into account cases, that is, plans created to solve similar problems in the past. Bajo et al. (2009) use case-based planning for the purpose of creating efficient working schedules in hospital environments. Fraile et al. (2013) employ case-based reasoning together with a belief, desire, and intentions model to solve new planning problems by adapting solutions of previous similar problems, and to learn by building upon initial knowledge. Case-based planning can be useful for ubiquitous computing due to the following reasons (Bajo et al. 2009, Fraile et al. 2013).

- *Learning ability* – case-based planning can handle dynamic environments due to its ability to learn from initial knowledge and past experiences.

- *Adaptive capacity* – the learning ability together with planner's capability to interact autonomously with an environment provides the case-based planner with a large capacity for adaptation to the needs of the environment.
- *Improve planning* – learning and adaptive abilities enable case-based planners to increase their ability to solve problems over time.

Bajo et al. (2009), however, indicate that case-based reasoning can be "highly affected" by context changes. In addition to past experiences, the success of finding a plan depends on the changes that may happen at execution time, which in turn may lead to contingencies and replanning.

Bidot et al. (2011) as well as Yordanova (2011) find the hybrid between HTN planning and Partial-Order Causal-Link (POCL) planning advantageous over approaches employing only hierarchical planning. Their hybrid planning is "more flexible" and does not require additional control knowledge. Moreover, the hybrid approach is "particularly advantageous" for real-world planning problems as it allows "to easily encode and efficiently deal with procedural knowledge" supported by the methods and task networks in HTN planning, and to reason about causal dependencies between tasks provided by the POCL planning. Milani and Poggioni (2007) use mixed integer programming (MIP) solver to search for a solution of a planning problem. The planning problem is first encoded as a plan graph, then the plan-graph relationships are translated into a mixed logical and numerical formula, and finally the formula is converted to a set of mixed integer linear programming constraints.

Several primary studies model only a planning problem and use state-of-the-art planners to search for a solution. A graph-based planner is employed in (Heider 2003, Marquardt and Uhrmacher 2009a), a heuristic-based planner is used in (Yordanova 2011, Heider 2003), a temporal planner in (Kotsovinos and Vukovic 2005, Vukovic et al. 2007), and a partial-order planner in (Heider 2003). Ranganathan and Campbell (2004) adopt a hybrid between graph-based planning and SAT-based planning. Masellis et al. (2010) mix planning with programming, while Mastrogiovanni et al. (2010) propose affordance-based planning.

Planning problems

The focus of the *Planning problems* class is to discover whether the primary studies define explicitly the planning problems they aim to solve. Less than half primary studies are precise about the problem they are trying to solve within ubiquitous computing. In several studies, we can extrapolate the planning problem inspecting the input provided to a given algorithm. A planning technique practically requires as input an explicit description of a problem for which a solution is to be synthes-

ised. Same number of studies provide only informative descriptions of what automated planning consists without an explicit suggestion of the potential for using planning in the particular environment. A few studies only make a reference to AI planning and mention its use superficially.

Problem representations

The *Problem representations* class is concerned with the ways to represent planning problems. We discuss two ways that we identify during the qualitative analysis, namely classical and state-variable representation. Both representation models are equivalent in expressive power, meaning that a planning problem represented in one representation can also be encoded using the other representation (Ghallab et al. 2004).

Classical representations The class of *Classical representation* encapsulates approaches in which the state of the environment is a set of ground logical atoms that can be true or false. Further, the actions are expressions that specify which logical atoms should be in the state so that the action is applicable, and which logical atoms should change their values after the action application. There are several primary studies we are able to identify that take advantage of this classical representation.

State-variable representations The class of *State-variable representation* encompasses approaches that represent a state as a set of values of a finite set of state variables. Actions represent functions that change the values of those variables. A few primary studies employ the state-variable representation. For studies based on constraint satisfaction (Kaldeli et al. 2012, Courtemanche et al. 2008), state variables that range over finite domains is a natural representation. Pairs of variables and values are incorporated as well in (Pajares Ferrando and Onaindia 2013, Cirillo et al. 2012, Harrington and Cahill 2011), while Grześ et al. (2014) take rather classical approach and assume that all variables are Boolean only. Milani and Poggioni (2007) define a state as a pair of a set of logical atoms and a set of state variables (*cf.* numerical fluents). As for the actions, some studies represent actions as Boolean variables (Kaldeli et al. 2012, Courtemanche et al. 2008, Milani and Poggioni 2007), while preconditions and effects are encoded as constrains on the state variables (Kaldeli et al. 2012), or as a system of linear inequalities (Milani and Poggioni 2007).

Modelling classes

Modelling focuses on approaches, constructs and languages used to define a planning problem. Modelling a planning problem includes definitions of a domain

model and problem. An approach to model a planning problem defines the way of creating the domain model and problem, constructs define the expressions, and languages define the syntax used to create the planning problem.

Domain models

The class of *Domain models* deals with the process of defining domain knowledge for the purpose of solving planning problems. The term 'domain model' denotes a representation that portrays behaviours as found in the real domain, and provides semantics for the constructs in the domain (McCluskey 2002). Given only a planning theory, it is tedious and often impractical for designer to manually define planning domain models. A more practical approach would be to use tools that support the engineering of domain knowledge. Grześ et al. (2014) propose a probabilistic relational model to knowledge engineering of planning problems. A designer, possibly a domain expert, uses standard database tools, such as forms and Web interfaces, to perform a "psychological IU analysis",[4] and to populate a database with the results. Based on this probabilistic relational model, a POMDP specification is automatically generated (recall that this represents a planning problem). What is particularly interesting is that the domain designer does not have to be necessarily aware that the population of the database in practice represents a planning problem. On the other hand, Ortiz et al. (2013) propose an approach that does not require explicit inputs from people to generate domain models. The approach segments sensor time series in order to recognise actions performed by users, and states produced by such actions. Preconditions and effects of actions are learned from those sensor readings. Using this information, a domain model represented in PDDL is automatically generated.

Two studies provide simplified ways to generate domain knowledge as compared to the studies just discussed. Hidalgo et al. (2011) realise a tool that supports modelling of knowledge that consists of skills and experiences gathered from human experts in solving known problems. We propose a domain modeller that provides intuitive guidance to users for creation, viewing and modification of domain knowledge (see Section 6.3.2). The tool abstracts the way of modelling the domain and verification of the correctness of the knowledge entered with respect to the syntax of the input language of the supported planning system.

Problem definitions

The class of *Problem definitions* is concerned with the process of generating and composing a planning problem. In this class, we assume that a domain model is

[4] A method for transcoding interactions relevant to fulfil a specific task.

manually encoded, but it may be automatically translated, if necessary, into a representation understandable to the respective planning system. The other components of a planning problem, the initial state and goal, are generated automatically from the current state of the environment. Domain models are usually stored in some knowledge base and queried by the planning system upon its initialisation or when necessary (Bidot et al. 2011, Hidalgo et al. 2011, Kaldeli et al. 2012, Sánchez-Garzón et al. 2012, Fraile et al. 2013, Ha et al. 2005, Madkour et al. 2013, Georgievski et al. 2013, Jih, Hsu, Lee and Chen 2007). Domain models are enriched with additional semantic annotations needed during the planning process (Vukovic et al. 2007, Kaldeli et al. 2012, Madkour et al. 2013), or after planning and during the instantiation of a plan (Bidot et al. 2011).

The current values of environment objects are supplied to planning systems by other context-aware components (Ranganathan and Campbell 2004, Ding et al. 2006, Hidalgo et al. 2011, Kaldeli et al. 2012, Sánchez-Garzón et al. 2012, Fraile et al. 2013, Heider 2003, Georgievski et al. 2013). The initial state, which is represented in a standardised form acceptable by the respective planner, is automatically generated from such context information. Once generated, the initial state, which can be further maintained by the planning system, may automatically and dynamically incorporate future context changes (Kaldeli et al. 2012, Courtemanche et al. 2008). Finally, the goal is generated automatically from the request coming from a human (Ranganathan and Campbell 2004, Kotsovinos and Vukovic 2005, Ding et al. 2006, Bidot et al. 2011, Kaldeli et al. 2012, Ha et al. 2005), or another component that senses and reasons over the changes in the environment (Carolis and Cozzolongo 2007, Kaldeli et al. 2012, Courtemanche et al. 2008, Georgievski et al. 2013).

Expressiveness constructs

The class of *Expressiveness constructs* defines a critical aspect of planning. It refers to the required or preferred expressive power of a planning model adopted to represent a ubiquitous computing environment. Our vision is to have AI planning able to cover a wide spectrum of properties of ubiquitous computing. As a result of the qualitative analysis, we identify the following collection of expressiveness constructs suggested by several primary studies as needed to capture the semantics of these environments.

- *Conditional effects* are needed to efficiently coordinate environments and to provide a compact representation of device semantics (Kotsovinos and Vukovic 2005, Heider 2003), and they are also required to solve problems that involve moving objects (*e.g.,* Tars possesses some object) (Marquardt and Uhrmacher 2009a).

- *Multi-type elements* can be used to reduce the number of actions needed to be modelled. For instance, two actions, one for type object and another for type human, can be modelled as one if multi-typing is supported (Yordanova 2011).
- *Numeric-valued fluents* are common in ubiquitous computing environments (Kaldeli et al. 2012). The fluents are used to model variables with large domains, such as the temperature measure.
- *Extended goals* are desirable and "well suited" for adaptive and user-centric environments as ubiquitous computing environments are (Vukovic et al. 2007, Kaldeli et al. 2012).
- *Disjunction in preconditions* is particularly useful for a compact representation of device operations (Heider 2003).
- *Universal quantification in preconditions and effects* enables to represent actions that can cover an arbitrary number of objects, such as lamps, screens, windows, *etc.* This construct appears to be convenient in ubiquitous computing environments due to their constant evolution and dynamic extension (Heider 2003).
- *Time and resource constraints* can be useful in many cases (Vukovic et al. 2007, Heider 2003, Marquardt and Uhrmacher 2009a). We cover temporal properties in Section 2.3.1. On the other hand, discrete and continuous resources are identified as important for specific domains (Heider 2003).
- *Axioms* are a useful way to separate the domain-specific knowledge from the semantics of actions (Marquardt and Uhrmacher 2009a). For instance, an axiom can be used to derive that the brightness of a TV increases when the brightness of the surrounding environment decreases.

Languages

The *Languages* class focuses on the syntax used to express the physical properties and specific knowledge of ubiquitous computing environments. In particular, this class explores the modelling languages employed to define a domain model, including actions and domain-specific knowledge, such as compound tasks in hierarchical planning; the current state of an environment, such as predicates, variables or functions; and the goal given as an input to a respective planning system. The Language class could give suggestions about the preference of primary studies for planning languages used to define ubiquitous computing environments.

A majority of the primary studies use PDDL as a modelling syntax for their domain models, several studies make use of hierarchical-based languages,

such as SHOP2 (Nau et al. 2003) and Hierarchical Planning Definition Language (HPDL) (Castillo et al. 2006, Georgievski et al. 2013), two studies employ ADL (Pednault 1989), and the rest of them use non-planning modelling languages, such as Synchronized Multimedia Integration Language (SMIL) (Bulterman 2001), Scone (Santofimia et al. 2010), Extensible Markup Language (XML) (Bray et al. 2008), and OWL-S. As expected, a common way among the primary studies is to use the same modelling language for the state and goal as for the domain model.

Monitoring and recovery

The *Monitoring and recovery* class is concerned with the way planning systems deal with uncertainty in ubiquitous computing. Generally, two processes are defined and interleaved in order to circumvent deviations. The first one observes the changes in the state of an environment and the execution of plan actions. The second process handles unexpected context information and environment-specific contingencies that occur during the execution of the plan.

The *monitoring process* may consist of two tasks, namely sensing and execution monitoring. *Sensing* observes and provides an up-to-date view of the current state of an environment at planning time and/or execution time. Qasem et al. (2004) perform sensing by using local closed-world statements and the concept of "source relevance" to search for an appropriate information source and to update the knowledge base whenever there is insufficient information to validate conditions. In contrast to traditional approaches, the authors avoid using planning actions to model the sensing due to the fact that the search space is reduced as a consequence of the reduced number of possible actions; and each type of (query to) an information source should be modelled as a separate planning action, which may be impractical considering the complexity of ubiquitous computing environments. Similarly, sensing tasks are not considered as planning actions in (Kaldeli et al. 2012, Rocco et al. 2014). Instead, the values that sensing tasks provide, either periodically or when some condition is detected in the environment, are automatically updated and incorporated in the constraint network by adding or removing constraints. Interestingly, Kaldeli et al. (2012) provide support for sensing in goals through the use of the `find_out` construct.

Execution monitoring executes plan actions, monitors and verifies that they are executed as expected. We show the most commons steps performed by the primary studies in Algorithm 1. The execution of actions involves low-level invocations (of device operations or services) in right order (Bidot et al. 2011, Kaldeli et al. 2012) and time (Kotsovinos and Vukovic 2005). Monitoring and verification is usually performed before and after action execution.

The *recovery process* may include several tasks, such as precondition delay, ac-

Algorithm 1 Execution monitoring

Input: Plan

1: **for** each action in plan **do**
2: Query the action to check its availability in the list of currently available actions (Ranganathan and Campbell 2004)
3: **if** action not available **then**
4: **Call** recovery process
5: Check the validity of preconditions of the action (Vukovic et al. 2007, Sánchez-Garzón et al. 2012, Rocco et al. 2014, Madkour et al. 2013)
6: **if** precondition cannot be satisfied **then**
7: **Call** recovery process ▷ Once all preconditions are satisfied, observe action effects
8: **else if** effects are unexpectedly satisfied **then** ▷ due some exogenous event
9: Avoid action execution (Kotsovinos and Vukovic 2005)
10: Execute the action and analyse its outcome (Vukovic et al. 2007, Ding et al. 2006, Sánchez-Garzón et al. 2012, Fraile et al. 2013, Rocco et al. 2014, Madkour et al. 2013)
11: **if** outcome is not expected **then**
12: **Call** recovery process
13: **end for**

tion retrying, action replacement, and replanning. If a precondition cannot be verified (due to unexpected context changes), it is delayed by inserting a temporal constraint and re-evaluating the precondition later (Rocco et al. 2014), or replanning is invoked (Vukovic et al. 2007, Bidot et al. 2011, Fraile et al. 2013, Madkour et al. 2013, Garro et al. 2008). If the outcome of action execution is a permanent failure, the plan is terminated and replanning for the same goal (Kaldeli et al. 2012, Ding et al. 2006) or a new goal state is invoked (Ranganathan and Campbell 2004, Kotsovinos and Vukovic 2005, Vukovic et al. 2007, Madkour et al. 2013). If the outcome represents an erroneous behaviour, such as disconnection or timeout, the action can be re-invoked (Ranganathan and Campbell 2004, Kaldeli et al. 2012) or replaced with another instance of the same type (Vukovic et al. 2007, Bidot et al. 2011, Madkour et al. 2013), and if a failure is observed again, replanning is invoked (Ranganathan and Campbell 2004, Kaldeli et al. 2012, Madkour et al. 2013).

Sánchez-Garzón et al. (2012) use the domain knowledge to handle contingencies such that, depending on the context arisen, several alternatives represented as applicability conditions of compound tasks are possible during replanning. In (Cirillo et al. 2012), the execution of the current plan is terminated and replanning is invoked

when the human behaviour, which is predicted before planning, has been changed during the plan execution. Finally, case-based planners naturally support replanning in the reuse stage (Bajo et al. 2009, Fraile et al. 2013). When some contingency happens, a new planning cycle is initiated taking into account already executed actions.

Replanning in ubiquitous computing is a computationally expensive task (Bidot et al. 2011, Rocco et al. 2014). However, the primary studies, which build abstract plans and whose services are bound to specific instances later in the system's life cycle, can avoid replanning for cases when services appear or disappear from the environment by using only rebinding of services to other instances (Vukovic et al. 2007, Bidot et al. 2011, Madkour et al. 2013). Moreover, Rocco et al. (2014) reduce the impact of replanning by using several strategies on the constraint network depending on the situations, such as temporal propagation, resource and state variable scheduling, or action application.

Life cycle

The class of *Life cycle* defines the phase of the life cycle of a ubiquitous computing system in which a planning system is executed. The choice of a phase in which planning is used generally depends on the type of decisions we want to make in the environment. A strategic decision would answer the question of what basic tasks are to be executed and in which order, while an operational decision would answer the question of what devices should execute the tasks (Bidot et al. 2011). At design time or compile time, planning can provide only a strategic decision, while both types of decisions can be supported if planning is used at the time of the system's run.

Design-time planning Planning at design time is used primarily to make strategic decisions and create a solution in the form of a plan with abstract actions. Once such abstract plan is found, it is handled by another component at runtime, that is, abstract actions are instantiated by device operations actually present in an environment (Bidot et al. 2011, Vukovic et al. 2007). Given a user goal and a set of services available in the environment, both studies create sequences with abstract services that cannot be directly invoked. At runtime, available service instances are discovered and used to bind abstract services.

Compile-time planning One justification for this approach is given in (Krüger et al. 2011). A necessary requirement might be to have a time-bounded system able to react in real time to every possible situation in the environment. Operations that

are computationally complex, such as planning, are therefore shifted to run at compile time. Compile-time planning may be useful if, for instance, a pre-generation of action sequences, or an early identification of modelling problems, such as deadlocks, is needed.

Runtime planning This approach involves selecting and synthesising devices or services during runtime. For instance, recipes are build and adapted at runtime in (Kotsovinos and Vukovic 2005), working schedules are created at execution time in (Bajo et al. 2009), composition is calculated by reasoning on the most up-to-date services at runtime in (Kaldeli et al. 2012), office adaptations are produced during runtime in (Georgievski et al. 2013), or a user is provided with solutions to planning problems in (Fraile et al. 2013, Marquardt and Uhrmacher 2009a). Runtime planning enables a real-time approach to planning, and encapsulates also cases where domain-specific knowledge is added to a planner during runtime, as found in (Marquardt et al. 2008).

Development classes

The class of *Development* defines the aspects of software design, implementation and integration of planning systems. This class includes the steps taken by the primary studies, ranging from the conception of planning systems through their manifestation as software products to the integration of the systems into ubiquitous computing architectures. We therefore divide the Development class into two subclasses, namely *Design and implementation* and *Integration architecture*.

Design and implementation

The class of *Design and implementation* deals with the architecture design used to develop planning systems, and the level of software development achieved for such systems. Let us discuss each aspect separately.

The analysis of primary studies with respect to the development of planning systems yields many similarities among the designs of these systems. Consequently, we identify a possibility for a common characterisation of a planning architecture suitable for ubiquitous computing. Figure 2.9 shows a component diagram of the architecture resulted from the design commonalities of several primary studies (Qasem et al. 2004, Ranganathan and Campbell 2004, Kotsovinos and Vukovic 2005, Vukovic et al. 2007, Bajo et al. 2009, Liang et al. 2010, Santofimia et al. 2010, Bidot et al. 2011, Hidalgo et al. 2011, Kaldeli et al. 2012, Song and Lee 2013, Sánchez-Garzón et al. 2012, Fraile et al. 2013, Ha et al. 2005, Rocco et al. 2014, Madkour et al. 2013, Georgievski et al. 2013, Jih, Hsu, Lee and Chen 2007). The architecture con-

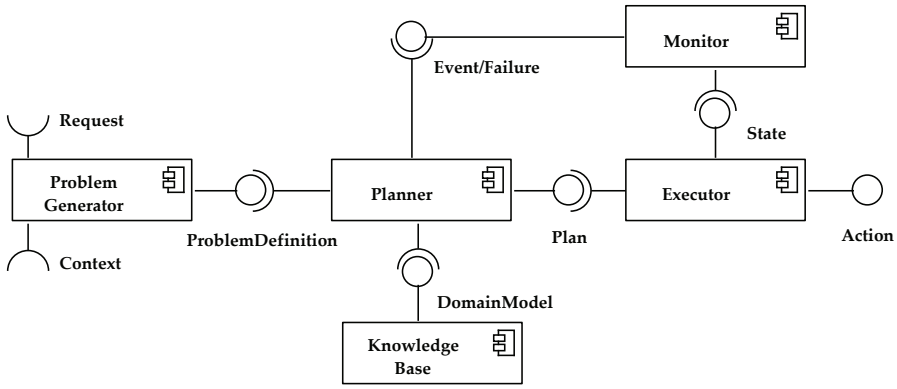


Figure 2.9: Component diagram of typical systems for planning in ubiquitous computing.

sists of five components. The Problem Generator component accepts two input ingredients: request, which describes the objective issued either by a user or by another component, and context, which represents the high-level information about the environment. The component generates a problem definition interpretable by the Planner component. Along with the problem definition, the Planner component, which implements a particular planning technique, requires a suitable domain model provided by the Knowledge Base component. Given these input ingredients, the Planner finds a solution plan, if one exists, and passes it to the Executor component. The Executor is responsible for the execution of each plan action in the environment. The execution of action is observed by the Monitor component. Upon deviations from the expected flow, the Monitor reacts accordingly either by repairing the situation arisen, or by invoking the Planner to search for a new solution.

The second aspect should provide insights into the maturity of the software solutions proposed by the primary studies. However, we identify only three meaningful pieces of information. First, the majority of primary studies implement prototype software of their proposal. That is, incomplete versions or only a few aspects of the proposal are provided. A working prototype is implemented in (Ranganathan and Campbell 2004, Kotsovinos and Vukovic 2005, Vukovic et al. 2007, Bajo et al. 2009, Hidalgo et al. 2011, Kaldeli et al. 2012, Sánchez-Garzón et al. 2012, Fraile et al. 2013, Ha et al. 2005, Rocco et al. 2014). Second, many primary studies employ or extend one or several state-of-the-art planners. Within the group of hierarchical planning, the majority of studies adopt some version of the SHOP planner (Nau et al. 1999). SHOP is extended in (Qasem et al. 2004), SHOP2 is used in (Ha et al. 2005, Marquardt et al. 2008), and JSHOP2 is employed in (Ding et al. 2006, Liang et al. 2010, Song and Lee 2013). Two studies (Hidalgo et al. 2011, Sánchez-

Garzón et al. 2012) use the SIADEX planner (Castillo et al. 2006), and Amigoni et al. (2005) uses a modified version of the NOAH planner (Sacerdoti 1975a). Furthermore, Cirillo et al. (2012) extend the PTLplan planner (Karlsson 2001), while the graph-based LPG planner (Gerevini and Serina 2002) is employed in (Heider 2003, Marquardt and Uhrmacher 2009a). The heuristic-based FF planner (Hoffmann and Nebel 2001) is extended in (Yordanova 2011), the heuristic-based planners Metric-FF (Hoffmann 2003) and MIPS (Edelkamp and Helmert 2001) are used in (Heider 2003); the temporal TLPlan planner (Bacchus and Kabanza 1996) is used in (Kotsovinos and Vukovic 2005, Vukovic et al. 2007), the partial-order UCPOP planner (Penberthy and Weld 1992) in (Heider 2003), and the Blackbox planner (Kautz and Selman 1999) is adopted in (Ranganathan and Campbell 2004).

Only a few studies implement new planners. Kaldeli et al. (2012) implement their approach in the RuGPlanner, Bajo et al. (2009) develop the case-based CBPMP planner, Pajares Ferrando and Onaindia (2013) build the CAMAP planning system, and we develop and use the **SH** planning system (Section 6.3).

Integration architectures

The class of *Integration architecture* defines and analyses paradigms upon which ubiquitous computing systems integrating a planning system are designed and implemented. We opt for ubiquitous computing systems truly and entirely realised in real environments that would therefore require a standardisation in both, at the system level, and at the single-component level (Degeler et al. 2013). A ubiquitous computing architecture must consider *scalability* and *distribution* as two important design characteristics currently challenges for many ubiquitous computing systems. Inherently, these two requirements are challenging for all architectural entities, including the planning one. A planning system needs to communicate and cooperate with other entities of the ubiquitous system. Communication and cooperation prompt for *interoperability* as another requirement for planning systems. The rapid evolution of technology implies that systems already deployed in ubiquitous computing environments will be soon outdated compared to the most recent ubiquitous systems. This implication represents a requirement to planning systems: they need to be able to catch up with new advances and to easily *evolve*. Additionally, the complexity of their adaptation to different types of ubiquitous computing environments must be taken into account. *Reusability* of planning systems gives ubiquitous technological solutions an opportunity for a wider use, and also a possibility for solutions to grow in power and complexity.

The qualitative analysis of primary studies resulted in three architecture paradigms, namely *Multi-agent systems*, *Modular architectures*, and *Service-oriented architectures*.

Multi-agent systems A multi-agent system is one that consists of a collection of agents. Each agent is a computer system that first, is capable to exhibit to some extent an autonomous behaviour – to decide what to do so as to satisfy some objectives, and second, is capable to interact with other agents – to exchange messages through a network, but also to engage in sort of social activities (Wooldridge 2009). A successful interaction depends on the abilities of agents to cooperate, coordinate and negotiate with each other. Some primary studies indicate that multi-agent systems are a "natural" (Amigoni et al. 2005) and "relevant" (Bajo et al. 2009) paradigm for ubiquitous computing environments, and that they "facilitate" the development of such environments (Fraile et al. 2013). A common approach taken by the primary studies to the design and implementation of multi-agent systems for ubiquitous computing environments is to consider environment devices or objects as agents, software components as agents, and a single planning agent (Amigoni et al. 2005, Bajo et al. 2009, Santofimia et al. 2010, Fraile et al. 2013, Jih, Hsu, Lee and Chen 2007). Amigoni et al. (2005) consider devices as simple agents that neither support context reasoning nor participate in distributed planning. The other studies report similar designs: the main assumption is that, besides device agents, other agents can only extract and provide context information and domain knowledge to the planning agent that has solely the reasoning capabilities to achieve the desired objectives. Pajares Ferrando and Onaindia (2013) take an approach closer to the core idea of the multi-agent paradigm, and in fact, employ multi-agent planning (Weerdt and Clement 2009) to search for a plan. Since ubiquitous computing environments have imperfect context information, a distribution of responsibilities, such as in the domain of health-care assistance, and heterogeneity of local context theories, it is required to have agents that are able to exchange and support their decisions, to interact with each other and to derive a joint plan as a solution to the problem. Several agents can thus be involved in the process of creating a plan according to their context information and reasoning.

Modular architectures A modular architecture is a design model in which a system consists of distinct modules that can be interconnected together. Modules represent a separation of functionality of a system into independent and logically bound concerns. Three primary studies indicate the use of a planning module within a modular architecture (Courtemanche et al. 2008, Sánchez-Garzón et al. 2012, Garro et al. 2008). Various modules, such as a diagnosis module (Courtemanche et al. 2008) or actuator module (Garro et al. 2008), are incorporated to support the planning module. When connected together, the modules form an executable system. Modules communicate through interfaces, which describe objects required and provided by a module. For instance, in (Courtemanche et al. 2008), all modules

communicate using XML messages. The use of interfaces is a small step towards standardisation. A modular architecture supports reusability when new applications are built by reusing and modifying existing modules (Sánchez-Garzón et al. 2012). Unfortunately, no further technicalities on modules can be extracted from the primary studies.

Service-oriented architectures Service-oriented architectures (SOAs) are an architectural model that enhances efficiency, agility, evolution and productivity by considering services as a primary way through which logic is represented. In this context, services are used to represent the functionalities of devices to sense and act in the environment in which they are deployed (Qasem et al. 2004, Vukovic et al. 2007, Masellis et al. 2010, Santofimia et al. 2010, Bidot et al. 2011, Kaldeli et al. 2012, Song and Lee 2013, Ha et al. 2005, Marquardt et al. 2008, Madkour et al. 2013, Marquardt and Uhrmacher 2009a, Jih, Hsu, Lee and Chen 2007). Services are also used to design, implement and execute ubiquitous computing systems (Kaldeli et al. 2012, Georgievski et al. 2013). The first type of services are ubiquitous computing services, and the second one are application services (see Section 1.4).

2.3.3 Interpretation

The objective of the *Interpretation* class is to provide insights into practical aspects of the theories and solutions proposed in the primary studies. Practical aspects may refer to the approaches taken to understand better and demonstrate the complexity and applicability of theories. Further, practical aspects may include a technical and qualitative evaluation of solutions, and may consider an examination of user acceptance and satisfaction of these theories and solutions. We therefore analyse the primary studies with respect to four subclasses of the Interpretation class, namely *Demonstrations*, *Quantitative evaluation*, *Qualitative evaluation*, and *User satisfaction*. We provide the classification of primary studies with respect to these classes in Table 2.4.

Demonstrations

The *Demonstrations* class describes the ways used to illustrate the complexity, and to evaluate the feasibility of a particular planning technique. While this class provides details on the approaches taken to demonstrate a proposed theory, it may also indicate the distance of a proposal from a real situation. In addition, this class points out the most common way of demonstration used by the primary studies. The Demonstrations class has three subclasses, namely *Scenarios*, *Examples*, and *Real-life settings*.

Table 2.4: Classification of the primary studies with respect to the Interpretation classes.

Study	Demonstrations			Quantitative eval.	Qualitative eval.	Usability eval.	Study	Demonstrations			Quantitative eval.	Qualitative eval.	Usability eval.
	Scenarios	Examples	Real-life settings					Scenarios	Examples	Real-life settings			
S1							S20		✘		✘		
S2	✘	✘		✘			S21	✘	✘		✘	✘	
S3	✘	✘		✘			S22		✘	✘		✘	
S4	✘	✘					S23		✘	✘			
S5	✘	✘					S24		✘				
S6	✘	✘		✘	✘		S25		✘	✘			
S7	✘	✘					S26		✘				
S8		✘			✘		S27		✘				
S9			✘	✘	✘	✘	S28	✘		✘			
S10		✘					S29	✘	✘	✘			
S11		✘					S30	✘	✘				
S12		✘		✘			S31		✘				
S13		✘					S32		✘		✘	✘	
S14	✘			✘			S34		✘				
S15		✘					S33		✘		✘	✘	
S16			✘			✘	S35	✘	✘		✘		
S17							S36	✘	✘				
S18	✘	✘		✘		✘	S37						
S19	✘	✘											

Scenarios

A *scenario* provides a synoptic description of people's and system's actions and events in a ubiquitous computing environment. It is a powerful illustration of the complexity of problems and their solutions. Scenarios should help people have a sufficiently wide view about the proposed idea so as to avoid missing important attributes of corresponding planning problems (Alexander and Maiden 2004). Nevertheless, scenarios are the starting point of all modelling and design (Sutcliffe 2003).

Almost half of the primary studies use scenarios to introduce and illustrate the (planning) problem of interest. Most of these scenarios are from the domain of smart homes, while the rest are from the domain of smart offices, smart hospitals and infotainment systems (information-providing systems). Some scenarios focus on a very specific use case and solve a single planning problem (Amigoni et al. 2005, Ding et al. 2006, Ranganathan and Campbell 2004, Carolis and Cozzolongo 2007,

Marquardt and Uhrmacher 2009a), while other describe various use cases and address multiple problems (Kotsovinos and Vukovic 2005, Bidot et al. 2011, Kaldeli et al. 2012, Pajares Ferrando and Onaindia 2013, Cirillo et al. 2012, Rocco et al. 2014, Jih, Hsu, Lee and Chen 2007, Marquardt and Uhrmacher 2009a, Song and Lee 2013). All scenarios illustrate the characteristics of problems and solutions from the perspective of a person who is explicitly or implicitly involved.

Examples

When introducing an approach taken to address a particular problem within ubiquitous computing, it is essential for the understanding and applicability of the approach to be exemplified. An *example* supports and clarifies what is introduced and meant. Examples are the most common form of demonstration taken by the primary studies. We identify that examples can be descriptive (that is, included in the text), or examples can be represented in a chosen syntax. The latter may include excerpts from a state representation, a goal example, parts of domain knowledge, such as a single action and/or decomposition, and an example of a plan. We have only eleven examples of what a planning state may be represented as, a few more examples of what a goal is, and 22 examples of different types of actions. A plan is exemplified in fourteen primary studies.

Real-life settings

The class of *Real-life settings* offers a particularly interesting perspective as it provides details on the application of AI planning in actual ubiquitous computing environments. As shown in Table 2.4, only seven primary studies are tested in real environments. Four studies are deployed in homes (Fraile et al. 2013, Ha et al. 2005, Rocco et al. 2014, Cirillo et al. 2012), one study in a university laboratory (Sando and Hishiyama 2011), one study in a hospital (Bajo et al. 2009), and one study in a care facility (Grześ et al. 2014). Bajo et al. (2009) involves the largest number of participants, namely thirty patients and six nurses, while other studies involve ten people (Sando and Hishiyama 2011), seven persons with dementia (Grześ et al. 2014), one home occupant (Fraile et al. 2013), one person and one robot (Ha et al. 2005, Cirillo et al. 2012), and two robots (Rocco et al. 2014). We identify the duration of real-life experimentation only for three studies: three partial days in (Sando and Hishiyama 2011), five hours in (Cirillo et al. 2012), and three months in (Fraile et al. 2013). Almost all studies provide a clear description of the entities, such as devices and locations, used for the real-life experiment. Furthermore, only one study (Fraile et al. 2013) consults experts to create the domain knowledge for the experiment. Finally, the majority of the studies perform the real-life testing only for exploratory purposes.

Quantitative evaluation

The factual demonstration of proposed planning approaches is covered by the class of *Quantitative evaluation*. In fact, this class deals with the feasibility of an approach expressed through an evaluation of the performance of the adopted planning technique. The analysis of primary studies helps to define the following aspects of interest for a technical evaluation of planning for ubiquitous computing.

- *Technical configuration* refers to the configuration of the technical setting in which tests are conducted. Five primary studies (Kotsovinos and Vukovic 2005, Vukovic et al. 2007, Bidot et al. 2011, Kaldeli et al. 2012, Milani and Poggioni 2007) or about 40% of the classified studies reveal the technical setting in which the respective approach is deployed and tested.

- *Algorithmic configuration* deals with the configuration of the planning algorithm used to run the approach. The cases of a planning algorithm configured with different runtime properties must be explicitly noted as the results of the quantitative evaluation depend directly on those configurations. For instance, Kotsovinos and Vukovic (2005) run the TLPlan planner in the mode of breadth-first search without knowledge on search control, while Kaldeli et al. (2012) use a random branching strategy during constraint solving with halting the search after a maximum number of backtracks.

- *Problem specification and performance* involves the need for specification of a particular planning problem used to produce a concrete and suggested performance result of the approach. The computational complexity of a planning problem is usually a function of the number of domain actions and/or methods (Vukovic et al. 2007, Bidot et al. 2011, Kaldeli et al. 2012, Pajares Ferrando and Onaindia 2013, Marquardt and Uhrmacher 2009a), objects (Kotsovinos and Vukovic 2005, Vukovic et al. 2007, Mastrogiovanni et al. 2010, Bidot et al. 2011, Georgievski et al. 2013), variables (Kaldeli et al. 2012), rooms (Kaldeli et al. 2012), the room topology (Milani and Poggioni 2007), the complexity of the goal (Kaldeli et al. 2012, Milani and Poggioni 2007, Georgievski et al. 2013), *etc.* Some studies, such as (Kaldeli et al. 2012), report that the performance of a planner may not be affected by the number of domain actions as much as it can be affected by the structure of the domain itself and goal. The structure of the domain refers to the causal dependencies between actions. These settings define rather *ideal conditions* for the evaluation of planners. In addition to the evaluation in ideal circumstances, there is at least one more case to be considered that includes failure of an action or a service. The evaluation of

planners under *faulty conditions* is considered in several primary studies, such as (Ranganathan and Campbell 2004, Vukovic et al. 2007, Kaldeli et al. 2012).

- *Computational factors and scalability* focuses on factors that influence the computational efficiency of a planning technique, and include a worst-case analysis of the performance of planning systems with and without action contingencies. Influential factors relate to the aforementioned function arguments. The scalability of planning systems for ubiquitous computing environments can be defined with respect to two factors, namely the *size of a domain* and the *size of a solution plan*. In other words, these two factors influence the size of space that a planning system searches in. Planning problems of varying and increasing size of the search space can be created by (1) increasing the initial state (Kotsovinos and Vukovic 2005, Vukovic et al. 2007, Mastrogiovanni et al. 2010, Kaldeli et al. 2012, Pajares Ferrando and Onaindia 2013, Milani and Poggioni 2007, Georgievski et al. 2013), (2) increasing the size of a request (Vukovic et al. 2007, Kaldeli et al. 2012, Pajares Ferrando and Onaindia 2013, Milani and Poggioni 2007, Georgievski et al. 2013), and (3) increasing the number of domain actions (Kaldeli et al. 2012). In contrast to point (3), Milani and Poggioni (2007) show that in environments that support reactive devices, a higher number of reactive devices (that is, actions) can make the planning problem easier to solve.[5] Finally, the worst-case analysis can be performed by using randomly generated sets of planning problems. For instance, Mastrogiovanni et al. (2010) analyse a worst case by running 100 and 200 iterations of their algorithm over a varying number of randomly generated objects (*cf.* neurons), ranging from 1000 to 6000. The number of iterations is related to the length of the solution. That is, 100 iterations of the algorithm correspond to the longest sequence of 100 actions. The analysis shows that the complexity of the worst case is not linear in the number of objects.

Qualitative evaluation

The class of *Qualitative evaluation* is concerned with the quality of solutions produced. This type of evaluation might be rather subjective as it may be based primarily on opinions drawn from observations. During the analysis of the primary studies, we recognise generally two categories of qualitative evaluation, namely an evaluation may answer *how well a solution plan is created in relation to specific parameters*, and/or *how well a solution plan is created in comparison to the solutions of other approaches*.

[5] Reactive devices represent mandatory choices which results in elimination of many branching points in the search space.

Vukovic et al. (2007) analyse their approach in relation to four parameter groups, such as task specification, application behaviour specification and configuration, application execution, and unpredictability and failure recovery. Each group contains several parameters each of which takes a particular value. For instance, the parameter indicating a specification of goal that belongs to the task specification group can take one of three values from a development perspective, namely manual, semi-automated, and automated. From a complexity perspective, the same parameter can take one of three values as well: easy, moderate, and difficult. Courtemanche et al. (2008) validate the quality of their approach by analysing the duration of plans proposed to a user in relation to their complexity, where the complexity is represented by the number of parallel actions that need to be carried out by the user. That is, the duration of a plan is reduced by decreasing the latency between plan actions due to the parallelism. Bajo et al. (2009) evaluate generated plans in relation to the time spent by nurses to supervise and control patients, the time spent for false alarms and for direct patient care, and in relation to security. The planning system may take care of some nurse's tasks which means more time for the nurse to carry out other (more important) tasks. The success or efficiency of a plan is evaluated based on the results obtained for each plan action and the feedback provided by the nurse that carried out the plan. Moreover, the authors use the relation between the average number of cases retrieved to solve a planning problem and the average number of replanning runs required to understand the behaviour of the system. Pajares Ferrando and Onaindia (2013) validate the quality of plans in relation to two parameters. The first one is the cost of a plan, assuming that actions are associated with a non-negative value, while the second one is the number of time steps of the plan, assuming that at each time step several actions can be executed in parallel. Fraile et al. (2013) evaluate the "advance in a plan" in relation to the sum of the "advances" achieved for each of the actions in the plan. Milani and Poggioni (2007) provide a qualitative analysis of plans in relation to a set of reasoning patterns of their planner. They distinguish four patterns, namely neglection, avoidance, exploitation, and prevention. In the neglection pattern, the planner does not take the presence of reactive device into account. This attitude has neither positive nor negative influence on the plan, but it has a side effect of an action not included in the solution. In the avoidance pattern, the planner inserts actions in the plan so as to avoid triggering of specific reactive devices which may prevent reaching the goal. In the exploitation pattern, the planner executes actions whose effect is to trigger specific reactive devices. In the prevention pattern, the planner intentionally activates reactive devices and prevents situations in which the search space would be infeasible.

Several primary studies compare their approaches with others qualitat-

ively. Vukovic et al. (2007) use the aforementioned parameter groups to compare their approach with a legacy application framework. The authors assume that such framework has fixed contextual dependencies, uses a "traditional application development strategy", and adopts "available application design toolkits and context middleware solutions". The way of assigning values for each parameter to both approaches is undefined. Pajares Ferrando and Onaindia (2013) use the two parameters (cost and number of time steps) to compare their approach with a traditional multi-agent planning system (with no argumentation mechanism for reasoning about context information). Fraile et al. (2013) use the "advances" of tasks to compare their system before and after its application. For instance, if the task is "control of medication intake", then a sensor is used to weigh the pill container and to inform the system about the medication intake. If the weight stays the same when a medication should have been taken, then there is an anomaly (an anomaly is measured in relation to the number of episodes of the abnormal patient behaviour).

Usability evaluation

The ISO 9241-11 standard defines usability as "the extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use".[6] Usability testing provides a fundamental method to evaluate such extent of products and systems (Wichansky 2000). It is also acknowledged that usability testing should be an essential phase in the development of ubiquitous computing environments (Kim et al. 2003). Therefore, the class of *Usability evaluation* deals with testing and evaluating the usability of planning systems particularly adopted in ubiquitous computing. Table 2.4 shows that there is a limited number of primary studies that take usability into account. This situation presents a clear gap in the understanding of usability requirements for planning systems, and on how to systematically evaluate usability. Our ambition is not to fill this gap, but to provide initial guidelines based on practices in primary studies. The basic steps of usability evaluation methodology could be as follows.

- *Determine users*. A planning system may have many distinct user groups such that each group may have its own goals with varying levels of effectiveness, efficiency and satisfaction. For instance, Kaldeli et al. (2012) identify two user groups with contrasting characteristics, a group of elderly and disabled people, and a group of young people who are technologically savvy. Bajo et al. (2009) identify nurses as a targeted group of users, while Sando and Hishiyama (2011) do not specify the users of focus.

[6] `http://en.wikipedia.org/wiki/ISO_9241`

- *Determine user goals*. Determining the user goals is a rather difficult step as it is unclear how to select the user goals important for a given problem. Kaldeli et al. (2012) use the term 'dimensions' to define the focus of their interest over users. We describe the following user goals based on the dimensions provided in (Kaldeli et al. 2012).
  - In ubiquitous computing environments, *acceptability* refers generally to the attitude of users towards the proposed solution and adopted technology. In (Kaldeli et al. 2012), acceptability comprises the attitude of users towards the importance of domotic technology, automation of tasks, and privacy. Sando and Hishiyama (2011) use several items, such as context-sensitive support, for the evaluation of the planner, and for each such item users score the level of importance.
  - *Learnability* refers to the process of gaining understanding about how to use a system. In (Kaldeli et al. 2012), learnability is represented by the amount of effort users have to make in order to understand the functionalities of the system, and to be able to use it.
  - *System effectiveness* is related to the satisfaction of users with the overall system. The system effectiveness can be an aggregate of several components, as in (Kaldeli et al. 2012) where the components are virtual environment effectiveness, user interface effectiveness, the support for complex goals, *etc.* Bajo et al. (2009) define two relations for the satisfaction of nurses with the system. The first relation is between the average satisfaction degree of a nurse with respect to the plan success and the number of retrieved cases to provide such plan. The second relation is between the number of retrieved cases, the average satisfaction degree of a nurse and the average number of replanning runs per plan. Sando and Hishiyama (2011) allow users to score their satisfaction level for each item in the evaluation form, and use a correlation coefficient between the level of importance and the level of satisfaction to evaluate the effectiveness of the planning system.
  - *Efficiency* refers to the time that a system takes to perform assigned tasks. In (Kaldeli et al. 2012), the efficiency is measured according to the user's assessment of the time required to complete simple operations and complex goals.
- *Determine the context of use*. In (Kaldeli et al. 2012), the context of use is determined by the "diverse requirements, abilities and technological knowledge" of targeted users within the project whose context is a smart home. In (Bajo et al.

2009), the context of use is determined by the profiles of nurses and patient needs, while in (Sando and Hishiyama 2011), the context of use is determined by the needed ingredients that users have to collect.

- *Determine the levels of importance, effectiveness, efficiency and satisfaction*. This is a challenging step as it requires determination of the 'right' levels, but also a crucial step as it defines the actual usability of a planning system (or a ubiquitous system in general). Kaldeli et al. (2012) use a scale from 0 to 4, while Sando and Hishiyama (2011) use levels from 0 to 5. Bajo et al. (2009) use an average satisfaction degree expressed in percentage.

2.4 Remarks

We observe that there are a number of interesting and challenging issues which remain still open. We are urging attention to focus and precision of defining future planning problems in ubiquitous computing. A study claiming to use planning should clearly define the planning problem being solved. We recognise a necessity for reporting more details on the translation of ubiquitous computing environments into planning problems, and on the actual representation of such problems. The use of ambiguous terms certainly leads to misunderstandings and misinterpretations of an approach. Further, preferences, spatial and temporal properties represent topics that are insufficiently investigated in the existing approaches. With respect to the handling of uncertainty, we find that a formalisation of the plan execution semantics is needed together with a sound and complete algorithm able to monitor ubiquitous computing environments and to perform valid plan repairs at execution time. We also point out that modelling domain knowledge automatically can drastically foster the understanding of ubiquitous computing environments, and generally, the use of planning for ubiquitous computing. There is space for analysis of the constructs needed to support expressivity, and the effect of their use in actual environments. Currently, little or nothing is known about this subject. Implementation-wise, we need planning systems that can be easily plugged into a complex ubiquitous computing system with little effort. Therefore, future planning systems should pay attention to their capabilities to interoperate with other software components, to be distributed, to scale, and to naturally support the evolution of ubiquitous computing. Finally, we need studies in which information of practical matters is reported. Well-defined quantitative, qualitative and usability evaluations are more than desired so as to better understand all dimensions that affect the use of planning for ubiquitous computing. Nevertheless, we really urge to apply planning to real ubiquitous computing environments.

Chapter 3

Model and complexity of planning for ubiquitous computing

Now imagine planning for ubiquitous computing as the target of observation. While we indeed identify an extensive set of classes and give all sorts of explanations, we cannot see the dots connecting everything and making the whole that achieves the (research) purpose.

In ubiquitous computing, one usually uses various kinds of models, such as environment models, planning models, architecture models, data models, and simulation models, to understand some problem at hand. Though each type of model deals with a specific aspect, all models have a common inception point with one purpose – to represent problems of the domain and their solutions. However, we have seen in Chapther 2 that there are many issues related to planning for ubiquitous computing that remain vaguely defined and leave space for misinterpretations. Some sort of a meta-model is needed that represents the inception point, that is, a model that organises all aspects of planning for ubiquitous computing and can be easily and clearly understood.

Let us go back to our target for observation and think about problems that planning is trying to solve in ubiquitous computing in terms of how difficult they are or the amount of resources they require. The knowledge about the complexity of planning in general helps in characterising the runtime behaviour of planning techniques for specific cases. More specifically, knowing the complexity of planning in specific domains gives an opportunity to outline the speed and length of plans generated by some techniques in those domains. Theoretical analysis may also expose some sources of hardness in a particular domain (Helmert 2003).

A wide range of planning domains have been suggested in ubiquitous computing, which makes the complexity analysis practically a difficult task. A better approach is to have a single and general ubiquitous computing planning domain in which we can analyse the complexity of planning. However, to the best of our knowledge, there are no domain definitions in the literature from which one can derive a general one. On the other hand, the descriptions, scenarios and actual environments in existing studies can help in generalising a ubiquitous computing do-

main.

In the following, we fix our target for observation by introducing a model specification, and formally defining a general ubiquitous computing planning domain. In order to develop a model specification of planning for ubiquitous computing, we use a widely used practice in Computer Science, called *conceptual modelling* (Embley and Thalheim 2011). In addition to providing a better overview of the domain of planning for ubiquitous computing, the model specification can orient the design and development of future systems due to model's commitment to clustering information according to its topic (Jeusfeld et al. 2009). It can be also used by the ubiquitous computing community to design enhanced and more intelligent solutions. Finally, having a well-defined ubiquitous computing planning domain provides means to characterise mathematically and discuss the complexity of planning problems in ubiquitous computing.

3.1 Conceptual modelling

Conceptual modelling is widely used in the field of Computer Science to elicit high-quality specifications of systems from some domain (also application domain or subject domain) (Thalheim 2010). It is defined as "the activity of formally describing some aspect of the physical and social world around us for purposes of understanding and communication" (Mylopoulos 1992). The structuring and inferential facilities supported by conceptual modelling are psychologically grounded because the resulting descriptions are intended for humans (in opposition to machines).

The description of situations from the real world should stand for the actual state of affairs of the domain under consideration (Guizzardi 2005). For example, if a planning problem is said to represent some problem from a ubiquitous computing environment, then this should reflect the actual state of affairs holding in reality. Abstractions of some part of the real world are created using concepts, which abstract representations of certain aspects of entities that exist in that domain (Guizzardi 2005, Jeusfeld et al. 2009). Concepts can be explicitly defined or implicitly assumed based on some common sense within the domain or a sub-field of Computer Science (Thalheim 2011). *Conceptualisation* gathers a set of concepts and relationships among them which are used to abstract away the state of affairs in the domain (Thalheim 2010, Guizzardi 2005). The abstraction of some part of reality according to a conceptualisation is called a *conceptual model* (Guizzardi 2005), *domain model* (Larman 2004), or *information model* (Jeusfeld et al. 2009). As abstract entities, conceptual models need to be represented in a concrete artefact in order to be further communicated and analysed. The representation of a conceptual model is called a *model specification*, which, in turn, is described using a *modelling language*.

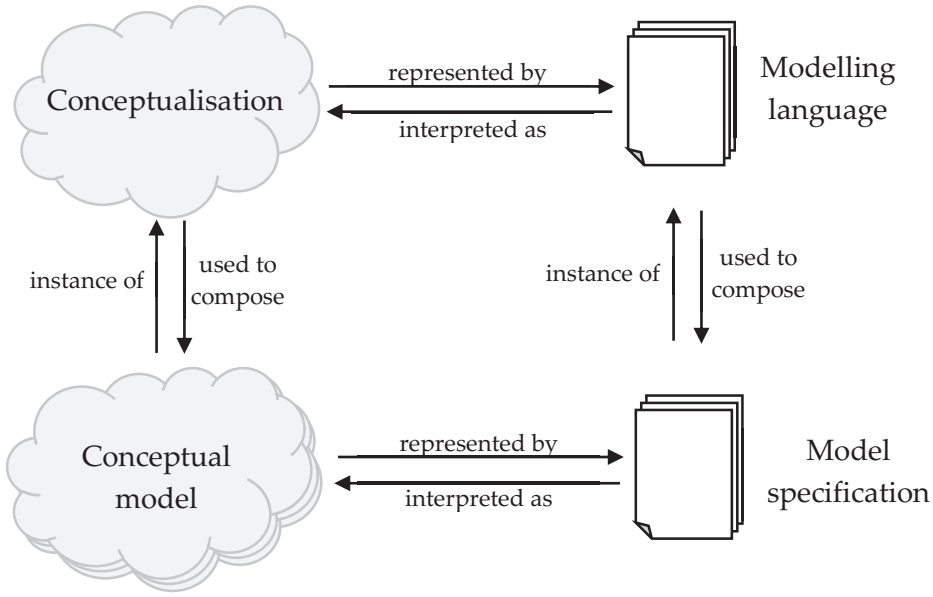


Figure 3.1: Relations between conceptualisation, conceptual model, model specification and modelling language (adopted from (Guizzardi 2005)).

The relation between conceptualisation, conceptual model, model specification, and modelling language is shown in Figure 3.1, which is adopted from (Guizzardi 2005).

In addition to concepts, conceptual models are constructed using structural relationships, such as aggregation, generalisation, exhibition, *etc.* (Thalheim 2010). For example, aggregation is used to characterise that a concept consists of some other concept, or generalisation enables to construct hierarchies of concepts based on inheritance assumptions.

Conceptual model specifications are mainly used as an intermediate artefact for system development (Thalheim 2011). In fact, a model specification is a description of the domain independent of specific system design and implementation choices (Guizzardi 2005). Model specifications are used by other stakeholders of the domain, such as software developers, learners, business users, and evaluators. That is, the specifications are used to support understanding, problem-solving, and communication among the stakeholders. Once there is a sufficient level of understanding and agreement about the domain, the conceptual model specification can be used as a blueprint for the further phases of the development process of the system.

Conceptual models use a modelling language as a carrier for specifications and

are restricted by the expressiveness of the carrier (Thalheim 2011). A common choice as a language for conceptual modelling is the Unified Modelling Language (UML), which deals with the construction of structural models (Booch et al. 2005). The UML fragment that is mostly used for conceptual modelling are class diagrams, *e.g.,* (Larman 2004, Guizzardi 2005, Walsh et al. 2007, Osis et al. 2007). Class diagrams are intended to represent the static structure of a domain. Generally, classes represent concepts and associations represent relationships between the concepts. While classes may contain attributes, method signatures are not allowed since methods are purely related to software design (Larman 2004).

The approach to conceptual modelling includes several activities (or acts), such as understanding, conceptualisation, abstraction, definition, construction, refinement, and evaluation (Thalheim 2010). The understanding act refers to the reasoning within the domain and results in preliminary data that can be used for the development of concepts. The conceptualisation act involves formulation of concepts and representation of those in the chosen language. The abstraction act outlines the main problem that must be supported by the system to be developed and abstract from unnecessary details. The definition act refers to defining the concepts used to develop the model specification in such a way that all ambiguities are removed. The construction act is the step that deals with the creation of the model specification by organising and linking concepts. The refinement act is an iterative step that improves the created specification by enriching or elaborating more the concepts and relationships. The final act is based on qualitative characteristics usually given in an abstract form for the entire model or parts of it.

3.2 Model specification

Following the preceding discussion, we here develop a model specification for the domain of planning for ubiquitous computing. For the understanding act, we can refer to the knowledge in Chapter 2. Thus, with the classes of properties of planning for ubiquitous computing in hand, we can focus on the conceptualisation. Since the classes represent generic properties or abstractions of aspects relevant to planning for ubiquitous computing, we map most of the classes to concepts. Considering the abstraction and definition acts, we derive concepts directly from the definitions of classes in Chapter 2. Table 3.1 shows the concepts and their definitions.

Table 3.1: Concepts derived from classes in Chapter 2.

Concept	Definition
Environment	Defined by the existence and interaction of artefacts, people, and robots in a part of the physical world. Objects, people and robots all have some relations with time and space, and the world itself is characterised by some degree of uncertainty. The environment takes on meaning and purpose only in the context of people's needs.
Behavioural input	People's desires according to which the environment should behave.
Request	The model of desires specified by people (or software components) that must be satisfied by the environment.
Declarative goal	A declarative description of a desired state of the environment.
Procedural goal	A set of procedures specifying how to accomplish a desired objective.
Preference	Individual desires towards the behaviour of the environment which should be satisfied as much as possible.
Behavioural output	The way in which the information that changes the environment is represented and produced.
Device operation	A functionality that some device can perform.
Human action	A behaviour to be performed by a person.
Robot action	A behaviour of a robot performed in order to achieve some goal.
Application service	A purposeful behaviour of an application.
Information service	A knowledgeable behaviour performed by collecting, managing and reasoning over data coming from distributed sources.
Physical property	A situation of a person, object or a place with respect to space and time.
Spatial property	A physical information that relates people and objects one another and with space.
Temporal property	The information used to organise the environment with respect to time.
Uncertainty	Unexpected events, changes and failures of behavioural outputs, and people behaviour over time.

Unexpected event	An event that happens in an exceptional and unpredicted situation.
Action contingency	The state of an operation (action or service) in which it does not work correctly during execution.
Partial observability	The imperfectness and incompleteness of information about the environment.
Planning technique	The technique used to realise planning.
Planning problem	The planning problem that needs to be solved.
Problem representation	The way in which a planning problem is represented.
Problem definition	The process of composing and generating a planning problem.
Expressiveness constructs	(Required or preferred) expressive power of planning.
Language	The syntax used to express the properties and other specific knowledge about the environment.
Monitoring and recovery	The way planning deals with uncertainty.
Interpretation	The practical aspects of the domain, including demonstration of applicability, technical and qualitative evaluation, and examination of user acceptance and satisfaction.
Demonstrations	The ways used to illustrate the complexity and to evaluate the feasibility of planning.
Quantitative evaluation	The feasibility of planning through an evaluation of the performance of the adopted planning technique.
Qualitative evaluation	The quality of plans produced by planning techniques by evaluating a plan in relation to some parameters or in comparison to plans created by other techniques.
Usability evaluation	The testing and evaluating the extent to which planning can be used by users to satisfy their desires.

Beside the concepts derived from the classes from Chapter 2, the domain of planning for ubiquitous computing involves several other relevant concepts discussed implicitly in the previous chapter and also mentioned in the definitions in Table 3.1. Looking at the definition of the Environment concept, one notices that artefacts, people and robots can be found in ubiquitous computing environments. Artefacts are objects embedded or abstractions present in the environments, such as devices, portables, applications, and information sources. Therefore, we define concepts for

Table 3.2: Additional concepts and their definitions.

Concept	Definition
Artefact	Objects embedded or abstractions present in the environment.
Device	A piece of equipment with limited capabilities to interact autonomously.
Portable	A small item that can be carried by a person or a robot.
Application	A piece of software that provides some functionalities.
Source	A document or software component that provides information.
Person	An individual within the environment.
Robot	An autonomous and intelligent to a certain degree device able to perform various tasks.
Goal	A description that represents the planning objective.
State	A description representing the current state of the environment.
(Planning) Domain	A description that represents behavioural output or other domain-specific knowledge.
Plan	A description representing the solution to some planning problem in terms of representatives from the behavioural output.

artefacts, people, and robots. In addition, Definition 2.3 describes that a typical planning problem consists of three entities, namely a goal, state, and domain (a set of actions). A solution to such a problem is called plan. We conceptualise these entities too. Table 3.2 shows these additional concepts and their definitions.

With the conceptualisation accomplished, we can now construct the specification of the conceptual model. We specify the concepts and their relationships in UML. Figure 3.2 shows the class diagram representing the conceptual model of planning for ubiquitous computing. With this specification, we intend to have a clear understanding of what a ubiquitous computing environment is composed of in order to model the planning problem determining the environment.

The Environment is composed of several concepts, namely the Behavioural input, Physical property, Behavioural output, Artefact, Person and Robot. All these concepts have a relationship to Environment specified as a composite association in UML. Each such an association has a multiplicity factor, which denotes the relation between the respective concept and its instances contained in the environment. For example, we may say that there is strictly more than one person in the environment,

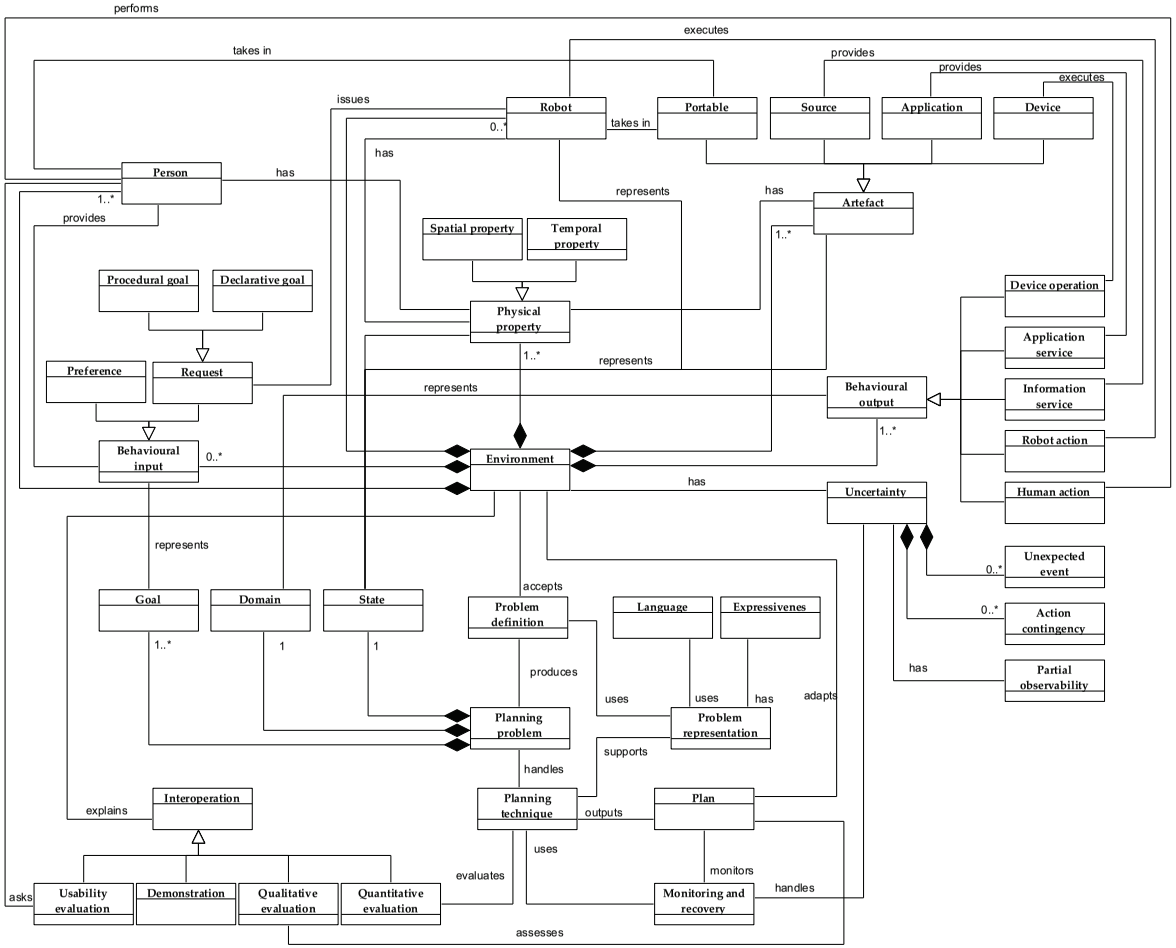


Figure 3.2: UML class diagram specifying the conceptual model of planning for ubiquitous computing.

but that the environment might not have a robot, or it may have several robots. The concepts that have specialised sub-concepts are connected with a generalisation relationship in UML. For example, Artefact is a general representation for Source, Application, Device, and Portable. An important dimension of the model specification covers the relationships which denote that two concepts are linked to each other or combined logically into some aggregation. These concept relationships are represented by associations in UML. For example, Artefact *has* Physical property, or Person *performs* Human action and *provides* Behavioural input.

Once the environment concepts and their relationships are clear, we can proceed further to the construction of the part related to planning, and to the establishment of its relation to the environment-related concepts. From Definition 2.3, we know that a planning problem is composed of a goal, state, and domain. Thus, the composite associations between the respective concepts. While the compositions of the domain and state have a multiplicity factor of one, this factor for the composition relationship of the goal is strictly greater than one (the goal may be a set of states, for example). The concept of Problem definition *produces* a Planning problem. It *uses* a Problem representation that *has* some Expressiveness and *uses* a modelling Language. A principal concept that aggregates all these concepts is the Planning technique. That is, a Planning technique *supports* some Problem representation, and *handles* a Problem problem. The Planning technique then *outputs* a Plan, if one exists. Moreover, the Planning technique *uses* some Monitoring and recovery technique that *handles* Uncertainty associated with the Environment. The main correlation between the Environment and a Planning problem is established by the Problem definition. The Problem definition *accepts* a description of the Environment and *produces* a Planning problem. More specifically, the Goal *represents* the Behavioural input, the Domain *represents* the Behavioural output and Preference, and the State *represents* the rest of concepts composing the Environment. The Plan, which solves a specific planning problem, *adapts* the Environment according to the Goal.

Having a correlation between planning and a ubiquitous computing environment, we go further and supplement the model specification with concepts relevant to the use and application of planning. The main concept is represented by the Interpretation class. Generally, this concept *explains* the Environment. In fact, it represents a generalisation of the Usability evaluation, Demonstration, Quantitative evaluation, and Qualitative evaluation concepts. Usability evaluation *asks* people within the environment (that is, Person) questions and demonstrates the level of effectiveness of and satisfaction with planning in the ubiquitous computing environment. Qualitative evaluation *assesses* the quality of a Plan, and Quantitative evaluation *evaluates* the performance of the Planning technique.

There are several concepts, such as Unexpected event and Action contingency, that can be further associated with other concepts of the Environment. For example, Behavioural output *has* an Action contingency. We do not show these associations in order to keep the model specification clearer.

3.3 Complexity

Planning techniques are usually evaluated on a set of benchmark domains, where benchmarks reflect critical aspects of actual applications. However, the adaptation of applications for use as benchmarks usually involves inevitable and often drastic simplifications (Hoffmann et al. 2006). Even in such a case, to the best of our knowledge, there is no benchmark domain for applications in ubiquitous computing. The traditional, but unrealistic way of developing a benchmark is based on a bottom-up approach by which a domain is artificially created according to some imaginery scenario for the respective application. Another way is to use a top-down approach such that actual applications of planning techniques are transformed into an appropriate domain model (Hoffmann et al. 2006). This approach is possible only if such applications exist and enough details are available.

Beside the common practice to evaluate planning techniques on problems from benchmark collections, there is also research conducted on the theoretical knowledge about the complexity of planning techniques in general (Bäckström and Nebel 1995, Bylander 1994, Erol et al. 1994a, 1995), and planning in specific domains (Gupta and Nau 1992, Helmert 2003). It is known that the upper bound for the complexity of planning in all domains is **PSPACE**-complete, where the domains are assumed to be encoded in STRIPS (Erol et al. 1995). Such knowledge helps in characterising the worst-case behaviour of planning techniques, but also outlining the speed and length of plans generated by some techniques for specific planning domains. Theoretical analysis may also help in exposing sources of hardness in a particular domain (Helmert 2003). For example, let us assume that planning problems in a domain of smart home with only one robot can be solved in polynomial time, but with two or more robots the corresponding problem is **NP**-hard. Then, we may conclude that one source of hardness is the number of robots.

Since it is impossible to analyse the complexity of every planning domain in ubiquitous computing, it makes more sense to look at those domains from a broader perspective, such as a class. This will enable us to define and analyse a single and general ubiquitous computing planning problem whose specialisations represent problems in specific domains. In other words, we are interested in planning tasks as structures that are characteristic to these domains, and not as encodings in propositional logic.

In order to define a general ubiquitous computing planning domain, as a first step, we have to decide which tasks should be part of the domain. As suggested in (Helmert 2003), one way is to use available domain definitions to gather valid tasks. In our case, this is impossible because, to the best of our knowledge, there are no domain definitions available in the literature. The other way is to identify a set of planning tasks by analysing domain descriptions available in the literature. Thankfully, we can refer to the primary studies from Chapter 2 and look at those providing descriptions of domains, scenarios and actual environments.

3.3.1 Analysis of existing domains

For the primary studies we use for the extraction of descriptions of domains, scenarios and actual environments, we refer to (Georgievski and Aiello 2015b). As we examine the descriptions, we observe that there are commonalities, but also differences between the various cases of ubiquitous computing. Most cases share the following properties, which indeed correspond to the properties of classes and concepts discussed in previous sections:

- There is a set of predefined *locations*, which may be connected by *doors* in such a way that they form a *layout*. For example, a kitchen, living room, bedroom, and bathroom are connected by doors, such as a door leading from the living room to the kitchen, forming a home.

- There is a set of *controllables*, which are devices that can be controlled, Controllables can be in a particular *state* and embedded in some location. For example, a TV is located in the living room, and currently turned on.

- There is a set of *persons*, which can be at some location and move through the layout. For example, a person in a wheelchair moves from the kitchen to the living room.

- There is a set of *robots*, which can be at some location, move through the layout, or perform some simple job at some location. For example, a domestic robot cleans the kitchen.

- There is a set of *portables* which can be at some location, moved by a robot or person, or manipulated by a person. When manipulated, a portable may be transformed. For example, a medication is taken in or a set of ingredients are chopped by a person.

- There is a set of *sources*, such as various sensors, address books, calendars, *etc*.

- There is a set of *applications*, such as a word processor, e-mail client, navigation system, *etc*.
- There is a set of *outputs*, which can be provided by sources, generated by controllables, or manipulated by applications. For example, a film can be displayed on the TV, or a light level can be provided by a natural light sensor.
- The goal is to move portables to the respective *final locations*, to transform portables to *needed forms*, to set controllables in *desired states*, and to generate and manipulate *needed outputs*.

There are also differences in terms of the support some domains provide for certain features:

- A domain may or may not include controllables. In those domains that consider them, the number of controllables is arbitrary.
- All domains involve at least one person, in some the number of persons is arbitrary.
- There are domains without robots, some with one robot, and domains with at most two robots.
- Some robots are associated with capacity constraints in terms of the number of portables they can carry at the same time. For example, a robot's tray may have capacity two.
- A domain may or may not include sources. In those domains that consider them, the number of sources is arbitrary, the source are assumed to be available upon query, and the number of queries is unlimited.
- A domain may or may not consider applications.
- A domain may or may not involve outputs. If a domain deals with outputs, it must include controllables, sources, applications, or any combination of those.

There are some features that we do not consider in the definition of our ubiquitous computing domain. This is because they relate to outdoor scenarios, or the information provided for a certain feature is insufficient or ambiguous. We exclude external location points, such as a home address, airport, hospital address, *etc*., mobiles other than robots, such as ambulances, cars and helicopters, specific information, such as access control policies, the concept of preferences, such as a preference for meal, music preference, room preference, *etc.*, and the notion of time.

There are several sets of actions we extract from the descriptions.

- There is a set of actions for controllables, such as *switching actions*, for example, to turn on the TV, and *setting actions*, for example, to set the alarm or open the curtains.

- There is a set of actions for persons, such as *movement, taking and leaving actions*, for example to go to the kitchen, get chips, move back to the living room and leave the chips, and *handling actions*, for example, to chop an ingredient.

- There is a set of actions for robots, such as *movement, picking and dropping actions*, for example, to move to the laundry room and collect and deliver laundry to the bedroom, and *cleaning actions*, for example, to clean the kitchen.

- There is a set of actions for outputs, such as *acquiring actions*, for example, to get a contact from an address book, *sending actions*, for example, to send an e-mail, and *performing actions*, for example, to display a film on the TV.

3.3.2 Ubiquitous computing task and domain

As we have seen in Section 2.1, planning relies on the concept of state model, which is defined over a state space and associated with a single initial state, a non-empty set of goal states, and a set of actions that deterministically map each state to another one (see Definition 2.1). Then, solving a planning problem (or task) means finding a sequence of actions that maps a specified initial state into some goal state.

We first define a ubiquitous computing task, which is later mapped into a state model using a domain we call Ubiquitous computing.

3.1 Definition (Ubiquitous computing task). *A ubiquitous computing* **task** T_{UC} *is a 19-tuple* $\langle L, C, I, P, D, E, O, H, R, \lambda_0, \iota_0, \omega_0, cap, layout, P_G, \lambda_G, C_G, \iota_G, \omega_G \rangle$*, where*

- L *is a finite set of* **locations**,

- C *is a finite set of* **controllables** *such that each* $c \in C$ *has a state transition function* $\tau_c : I_c \to I_c$*, where* I_c *is a finite set of internal states for* c*. We denote the set of all* **controllable states** *as* I*.*

- P *is a finite set of* **portables**, *where a portable may have a transformation function* $\mu_p : P \to P$,

- D *is a finite set of* **available sources**,

- E *is a finite set of* **applications**,

- O *is a finite set of* **outputs**,

- H *is a finite set of* **persons**,
- R *is a finite set of* **robots**,
- $\lambda_0 : (C \cup P \cup H \cup R) \to L$ *is the* **initial location** *function,*
- $\iota_0 : C \to I$ *is the* **initial controllable state** *function,*
- $\omega_0 : ((C \times i_0) \cup D \cup E) \to O$ *is the* **initial output** *function,*
- $cap : R \to \mathbb{N}$ *is the* **robot capacity** *function,*
- $layout : (H \cup R) \to \mathbb{P}(L \times L)$ *is the* **layout** *function, which is symmetric and irreflexive,*
- $P_G \subseteq P$ *is the set of* **goal portables**,
- $\lambda_G : P_G \cup H \cup R \to L \cup H \cup R$ *is the* **goal location** *function,*
- $C_G \subseteq C$ *is the set of* **goal controllables**,
- $\iota_G : C_G \to I$ *is the* **goal controllable state** *function,*
- $\omega_G : ((C_G \times i_G) \cup D \cup E) \to O$ *is the* **goal output** *function.*

Sets $L, C, I, P, D, E, O, H, R$ are disjoint. Controllables, portables, persons and robots have a specified initial location. Controllables are static, thus they remain in their initial location. Persons and robots are initially unloaded, thus portables are not associated with them at the beginning. The capacity function bounds the number of portables a given robot can carry at the same time. The layout function specifies paths for each robot and person. The pairs $(L, layout(H))$ and $(L, layout(R))$ are undirected graphs for all $h \in H$ and $r \in R$, respectively.

Before we can define Ubiquitous computing, we have to formalise the concept of planning domain which we adopt from (Helmert 2003).

3.2 Definition (Planning domain). *A planning domain D is a function that maps words over some encoding language to (planning) state models. A word T that is part of the domain of D is called a planning task of D.*

We can now define Ubiquitous computing. We use the notation $f \oplus (a', b')$ for functional overloading, that is, function f' with $f'(a') = b'$ and $f'(a) = f(a)$ for all $a \neq a'$.

3.3 Definition (Ubiquitous computing domain). *Given a ubiquitous computing task T_{UC}, ubiquitous computing is a planning domain such that maps T_{UC} to a planning state model as follows.*

The set of states consists of triples (λ, ι, ω) of current location *function $\lambda : (H \cup R \cup P) \rightarrow L \cup R$, the* current controllable state *function $\iota : C \rightarrow I$, and the* current output *function $\omega : ((C \times \iota) \cup D \cup E) \rightarrow O$.*

The initial state is given by the initial location, initial controllable state, and initial output functions. The set of goal states consists of those states in which the current location of all goal portables matches their goal location, the current state and output of each goal controllable matches its goal controllable state and output, respectively.

The set of actions consists of four subsets of actions. The first subset is called controllable and consists of switching actions $turn_{c,i}$, which turn controllable c to state i, and setting actions $set_{c,i}$, which cause controllable c to reach state i. The second set of actions is called robotic and consists of movement actions $move_{r,l}$, which move robot r to location l, picking actions $pickup_{r,p}$, which cause robot r to get portable p, dropping actions $drop_p$, which cause portable p to be put down by the robot currently carrying it, and cleaning actions $clean_{r,l}$, which cause robot r to clean dust (that is, a portable) from location l. The third subset is called personal and consists of daily actions $handle_{h,p}$, which instruct person h to handle portable p, and $move_{h,l}$, which suggest person h to move to location l. The last subset is called informational and consists of acquiring actions $get_{d,o}$, which query source d for output o, sending actions $send_o$, which send output o, performing actions $perform_{c,o}$, which use controllable c to perform output o.

The action $turn_{c,i}$ is defined in all states (λ, ι, ω) such that $\lambda(c) = \lambda_0(c)$, $\iota(c) \in I_c$, and there exists $\tau_c(\iota(c)) = i$. Its application results in state $(\lambda, \iota' = \iota \oplus (c, i), \omega)$. Other actions of the controllable set are defined analogously.

The action $move_{r,l}$ is defined in all states (λ, ι, ω), where $(\lambda(r), l)$ is part of the layout of r. Its application results in state $(\lambda' = \lambda \oplus (r, l), \iota, \omega)$.

The action $pickup_{r,p}$ is defined in all states (λ, ι, ω) such that $\lambda(r) = \lambda(p)$ and $|\{p \in P \mid \lambda(p) = r\}| < cap(r)$. Its application results in state $(\lambda' = \lambda \oplus (p, r), \iota, \omega)$.

The action $drop_p$ is defined in all states (λ, ι, ω) such that $\lambda(p) \in R$. It results in state $(\lambda' = \lambda \oplus (p, \lambda(\lambda(p))), \iota, \omega)$.

The action $clean_{r,l}$ *is defined in all states* (λ, ι, ω) *such that* $\lambda(r) = l$*, for all* $h \in H$*,* $\lambda(r) \neq \lambda(h)$*, and there exists* $p \in P$ *such that* $\lambda(p) = l$*. Its application results in state* $(\lambda' = \lambda \oplus (p, r), \iota, \omega' = \omega \oplus (l, \lambda(p))$*.*

The actions $take_{h,p}$*,* $move_{h,l}$ *and* $leave_p$ *are defined analogously to robotic actions* $move_{r,p}$*,* $pickup_{r,p}$*, and* $drop_p$*, respectively. The action* $handle_{h,p}$ *is defined in all states* (λ, ι, ω) *such that* $\lambda(p) = h$ *and there exists* $\mu(p) \in P$*. Its application results in state* $(\lambda' = \lambda \oplus (h, \mu(p)), \iota, \omega)$*.*

The action $move_{h,l}$ *is defined in all states* (λ, ι, ω)*, where* $(\lambda(h), l)$ *is part of the layout of* h*. Its application results in state* $(\lambda' = \lambda \oplus (h, l), \iota, \omega)$*.*

The action $get_{m,o}$ *is defined in all states* (λ, ι, ω) *such that either* $m \in D$ *or* $m \in E$*. Its application results in state* $(\lambda, \iota, \omega' = \omega \oplus (m, o))$*.*

The action $send_{e,o}$ *is defined in all states* (λ, ι, ω)*. Its application results in state* $(\lambda, \iota, \omega' = \omega \oplus (e, \emptyset))$*.*

The action $perform_{c,o}$ *is defined in all states* (λ, ι, ω) *such that* $\lambda(c) = \lambda_0(c)$ *and* $\iota(c) = i$ *is the current state of* c*. Its application results in state* $(\lambda, \iota, \omega' = \omega \oplus ((c, i), o))$*.*

3.3.3 Results

Ubiquitous computing can be thought of as an infinite set of planning tasks. For such a planning domain, we are interested in two decision problems formally defined as follows.

3.4 Definition (PlanEx-UbiquitousComputing). *Let* T_{UC} *be a planning task from Ubiquitous computing. The plan existence problem is to decide whether there exists or not a sequence of actions that maps* $(\lambda_0, \iota_0, \omega_0)$ *into* $(\lambda_G, \iota_G, \omega_G)$*.*

3.5 Definition (PlanLen-UbiquitousComputing). *Let* T_{UC} *be a planning task from Ubiquitous computing and* $k \in \mathbb{N}$*. The plan length problem is to decide whether there exists or not a sequence of actions of length at most* k *that maps* $(\lambda_0, \iota_0, \omega_0)$ *into* $(\lambda_G, \iota_G, \omega_G)$*.*

We now demonstrate a new property for planning tasks in Ubiquitous computing, that is, the solutions to such tasks consist of a polynomial number of actions.

3.6 Theorem (Membership in **NP** for PlanLen-UbiquitousComputing). *A solvable Ubiquitous computing task* T_{UC} *has a solution of length at most* $p(\|T_{UC}\|)$*, where* p *is a polynomial.*

Proof. To get the length of a solution to a Ubiquitous computing planning task, we have to consider and analyse all circumstances under which each action from Ubiquitous computing is applicable and may therefore be part of the solution. Now, suppose that T_{UC} is a Ubiquitous computing task as defined in Definition 3.1 and has a solution. Also, assume that $\|T_{UC}\| = |L| + |C| + |I| + |P| + |D| + |E| + |O| + |H| + |R|$. We go through each set of actions separately. Taking the set of controllable actions, we get that switching or setting a controllable from one state to another requires one action per state, which bounds the number of controllable actions by $|C| \cdot |I|$, or $\mathcal{O}(\|T_{UC}\|^2)$. The discussion on the next set of actions, that is, robotic actions, can be supported by the complexity analysis of planning in Transport domains (Helmert 2003). Each portable needs to be moved to a final location at most once, which bounds the number of picking actions by $|L| \cdot |P|$, or $\mathcal{O}(\|T_{UC}\|^2)$, and analogously for the number of dropping actions. Cleaning of each location bounds the number of cleaning actions by $|L|$, or $\mathcal{O}(\|T_{UC}\|)$. A robot should not visit a given location twice in between two pickup and drop actions and between the first and after the last such action, which bounds the number of movement actions by $2(|L| \cdot |P| + 1) \cdot (|L| \cdot |R|) \cdot (|L| \cdot |R|)$, or $\mathcal{O}(\|T_{UC}\|^6)$.

We can apply a similar reasoning to the set of personal actions. If each portable needs to be moved by a person to a final location at most once and handled at most once, then the number of taking, leaving and handling actions is bound by $(|L| \cdot |P|) \cdot |P|$, or $\mathcal{O}(\|T_{UC}\|^3)$, each. The visitation of locations for the delivery or handling of portables is not necessarily limited as in the robot's case, thus the number of movement actions is bound by $|L| \cdot (|L| \cdot |P|) \cdot (|L| \cdot |H|) \cdot (|L| \cdot |H|)$, or $\mathcal{O}(\|T_{UC}\|^7)$.

Finally, considering the last set of actions, acquiring and sending an output requires at most one action each, which bounds the number of acquiring and sending actions by $|D| \cdot |O|$, and $|E| \cdot |O|$, respectively, or $\mathcal{O}(\|T_{UC}\|^2)$ each. Performing an output on some controllable requires the controllable to be in the state necessary for the output to be performed, which in turn requires one action at most. This bounds the number of performing actions by $|C| \cdot |O|$, or $\mathcal{O}(\|T_{UC}\|^2)$.

If we add the bounds together, we get a total upper bound of $\mathcal{O}(\|T_{UC}\|^7)$, and therefore $p(\|T_{UC}\|)$. □

Since we have showed that the solution to T_{UC} is of length $p(\|T_{UC}\|)$, where p is a fixed polynomial, then we can use a non-deterministic algorithm to guess and check the solution in polynomial time. We can therefore state the following corollary.

3.7 Corollary (Membership in **NP** for planning in Ubiquitous computing). *The plan existence problems in Ubiquitous computing are in* **NP**.

Proof. A Ubiquitous computing task T_{UC} has a solution if and only if the task has a solution of length $p(\|T_{UC}\|)$ given Theorem 3.6. Generating $p(\|T_{UC}\|)$ from T_{UC} is

a polynomial-time reduction (Garey and Johnson 1990). □

3.4 Summary

Since the main context of the thesis is focused on planning techniques applied to ubiquitous computing, we defined the concepts and specified a conceptual model of planning for ubiquitous computing. The main purpose of the model is to express the meaning of concepts used by the experts to discuss relevant problems in this domain, and to identify the correct relationships between the concepts. The model provides a high-level explanation of planning for ubiquitous computing and it is therefore independent of design and implementation choices. We also defined a general planning domain for ubiquitous computing. The complexity analysis provided the first results on the worst-case behaviour of planning techniques when solving planning problems in this domain. That is, we identified the upper limit of speed and length of plans. Further analyses may help in identifying the sources of hardness of planning problems in ubiquitous computing.

Chapter 4

Hierarchical planning revisited

With planning techniques, one is able to solve various world's problems computationally. We have seen that the simplest and classical form of planning requires an initial state of a ubiquitous computing environment, a goal state, and some environment's actions to realise a sequence of actions that, when executed in the initial state, lead to the goal state. While actions present simple transitions from a world state to another one, a very common structure we use to understand the world better is of a hierarchical nature. The ability of planning to represent and deal with hierarchies is supported by *Hierarchical Task Network (HTN) planning*, or *hierarchical planning*. Hierarchies encompass rich domain knowledge characterising the world, which makes HTN planning to be very useful, and also to perform well in real-world domains.

HTN planning breaks with the tradition of classical planning. The basic idea behind this technique includes an initial state description, a task network as an objective to be achieved, and domain knowledge consisting of networks of primitive and compound tasks. A task network represents a hierarchy of tasks each of which can be executed, if the task is primitive, or decomposed into refined subtasks. The planning process starts by decomposing the initial task network and continues until all compound tasks are decomposed, that is, a solution is found. The solution is a plan which equates to a set of primitive tasks applicable to the initial world state.

Beside being a tradition breaker, HTN planning appears to be controversial as well. The controversy lies in its requirement for well-conceived and well-structured domain knowledge. Such knowledge is likely to contain rich information and guidance on how to solve a planning problem, thus encoding more of the solution than was envisioned for classical planning techniques. This structured and rich knowledge gives a primary advantage to HTN planners in terms of speed and scalability when applied to real-world problems and compared to their counterparts in classical world.

The biggest contribution towards this kind of "popular" image of HTN planning has emerged after the proposal of the Simple Hierarchical Ordered Planner (SHOP) (Nau et al. 1999) and its successors. SHOP is an HTN-based planner that shows efficient performance even on complex problems, but at the expense of

providing well-written and possibly algorithmic-like domain knowledge. Several situations may confirm our observation, but the most well known is the disqualification of SHOP from the International Planning Competition (IPC) in 2000 (Bacchus 2001) with the reason that the domain knowledge was not well written so that the planner produced plans that were not solutions to the competition problems (Nau et al. 1999). Furthermore, the disqualification was followed by a dispute on whether providing such knowledge to a planner should be considered as "cheating" in the world of AI planning (Nau 2007).

SHOP's style of HTN planning was introduced by the end of 1990s, but HTN planning existed long before that. The initial idea of hierarchical planning was presented by the Nets of Action Hierarchies (NOAH) planner (Sacerdoti 1975a) in 1975. It was followed by a series of studies on practical implementations and theoretical contributions on HTN planning up until today. We believe that the fruitful ideas and scientific contribution of nearly 40 years must not be easily reduced to controversy and antagonism towards HTN planning. On the other hand, we are faced with a situation full of fuzziness in terms of difficulty to understand what kind of planning style other HTN planners perform, how it is achieved and implemented, what are the similarities and differences among these planners, and finally, what is their actual contribution to the creation of the overall and possibly objective image of HTN planning. The situation cannot be effortlessly clarified because the current literature on HTN planning, despite being very rich, reports little or nothing at all on any of these issues, especially in a consolidated form.

We aim to consolidate and synthesise a number of existing studies on HTN planning in a manner that will clarify, categorise and analyse HTN planners, and allow to make statements that are not merely based on contributions of a single HTN planner. We also hope to rectify the perception of HTN planning as being controversial and antagonistic in the AI planning community. This work nevertheless serves as an extensive evaluation of the current state of the art of HTN planning.

4.1 Methodology

We inspect HTN planning from three different perspectives. The first one focuses on theoretical models of HTN planning. The second perspective provides a clarification of different concepts related to the search space, and context for interpretation of HTN planners. The last perspective enables us to go deeper and beyond dry descriptions about HTN planners by considering a set of functional, non-functional, and formal properties of planners.

We make use of two inclusion criteria for planners and studies. The *inclusion criterion of planners* relies on the inspection of existing literature for suggestions

on HTN planners that have risen to some degree of prominence. For example, we accept the list of "best-known domain-independent HTN planning systems" as provided in (Ghallab et al. 2004). In addition to those five suggested planners, we include two more. The complete list of HTN planners participants in our study is the following one:[1]

- NOAH, the first HTN planner emerged in mid-1970s (Sacerdoti 1975b,a),
- Nonlin that appeared one year later (Tate 1976, 1977),
- System for Interactive Planning and Execution (SIPE) and SIPE-2 introduced in 1984 and 1990, respectively (Wilkins 1991),
- Open Planning Architecture (O-Plan) and its successor O-Plan2 in 1984 and 1989, respectively (Currie and Tate 1991, Tate, Drabble and Kirby 1994),
- Universal Method Composition Planner (UMCP) introduced in 1994 (Erol 1996),
- SHOP and its successor SHOP2 that appeared in 1999 and 2003, respectively (Nau et al. 1999, 2003), and
- SIADEX that emerged in 2005 (Castillo et al. 2005).

The *inclusion criterion of studies* relies on the theoretical contribution of a study with respect to HTN planning in general, and theoretical and practical issues of each chosen planner separately. The criterion is based on the coverage a study gives, which may include information that ranges from a general discussion of techniques and approaches, peculiar matters, such as task interactions and condition types, relevant to our concepts, to properties, such as domain authoring, expressiveness and competence, that may be a part of the analysis.

4.2 Models

While there are several attempts to formalise a model for HTN planning (Erol et al. 1994c, Nau et al. 1999, Ghallab et al. 2004, Geier and Bercher 2011), each defines hierarchical terms appropriately to its underlying theory. In order to provide a basic understanding of HTN planning, we take these existing theories and generalise a hierarchical planning model in which we keep definitions of the terms high level. Further and as needed, we provide specific definitions of the terms characteristic

[1] Henceforth, we refer only to the most recent version of each planner.

for the particular hierarchical model. The basic model also determines the focus of categorisation of hierarchical planning that we propose later.

The *HTN planning language* is a first-order language that contains several mutually disjoint sets of symbols. Three of the sets are the following: P is a finite set of predicate symbols, C is a finite set of constant symbols, and V is an infinite set of variable symbols. These sets define the basic constructs of a state, that is, predicates. A *predicate* consists of a predicate symbol $p \in P$, and a list of terms $\tau_1, \ldots, \tau_k$. A *term* τ is either a constant symbol $c \in C$, or a variable symbol $v \in V$. Each predicate can be true or false, and a predicate is *ground* if its terms contain no variable symbols. We denote the set of all predicates as Q.

We can now define the state with respect to the state model (see Definition 2.1. A state $s \in S$ is a set of ground predicates 2^Q in which the closed-world assumption is adopted.

Characteristic for HTN planning are the notions of primitive and compound tasks. A *primitive task* (or primitive name) as an expression $t_p(\tau)$, where $t_p \in T_p$ and T_p is a finite set of primitive symbols, and $\tau = \tau_1, \ldots, \tau_k$ are terms. Each primitive task is represented by a single operator defined similarly to the STRIPS operator.

4.1 Definition (Operator). *An* operator o *is a triple* $(p(o), pre(o), eff(o))$*, where* $p(o)$ *is a primitive task,* $pre(o) \in 2^Q$ *are preconditions,* $eff(o) \in 2^Q$ *are effects.*

The subsets $pre^+(o)$ *and* $pre^-(o)$ *denote positive and negative preconditions of* o*, respectively, and* $eff^-(o)$ *and* $eff^+(o)$ *are negative and positive effects of* o*, respectively.*

As in the STRIPS planning problem, a transition from a state to another one is accomplished by an instance of an operator whose preconditions are a logical consequence of the current state. That is, an operator o is *applicable* in state s, if $pre^+(o) \subseteq s$ and $pre^-(o) \cap s = \emptyset$. The application of o to s results in state $s[o] = (s \cup eff^+(o)) \setminus eff^-(o) = s'$.

What makes hierarchical planning different from classical planning and a unique planning technique is the domain-specific knowledge expressed through compound tasks. A *compound task* (or compound name) is an expression $t_c(\tau)$, where $t_c \in T_c$ and T_c is a finite set of compound symbols, and $\tau = \tau_1, \ldots, \tau_k$ are terms. We refer to the union of the sets of primitive and compound names as a set of task names T_n. The following two definitions are further complemented for the respective model of HTN planning.

4.2 Definition (Task network). *A* task network tn *is a pair* $\langle T, \psi \rangle$*, where* T *is a finite set of tasks, and* ψ *is a set of constraints.*

Constraints in ψ specify restrictions over T that must be satisfied during the planning process and by the solution. We refer to a task network over the set of

primitive tasks as a *primitive task network*. The set of all task networks over T_n is denoted as TN.

4.3 Definition (Method). *A* method m *is a pair* $\langle c(m), tn(m)\rangle$*, where* $c(m)$ *is a compound task, and* $tn(m)$ *is a task network.*

We can now define the problem in HTN planning.

4.4 Definition (HTN planning problem). *An* HTN planning problem $\mathcal{P}$ *is a tuple* $\langle Q, O, M, tn_0, s_0\rangle$*, where*

- Q *is a finite set of predicates,*
- O *is a finite set of operators,*
- M *is a finite set of methods,*
- tn_0 *is an initial task network,*
- s_0 *is the initial state.*

From this definition, we can understand another difference that the hierarchical model has with the classical planning model. Planning is no longer searching for a sequence of actions that maps an initial state into some goal state, but instead hierarchical planning searches for a sequence of actions that accomplishes the initial task network when applied to the initial state. As in classical planning, an operator sequence $o_1, \ldots, o_n$ is *applicable* in s if there is a sequence of states $s_0, \ldots, s_n$ (also called a *trajectory*) such that $s_0 = s$ and o_i is applicable in s_{i-1} and $s_{i-1}[o_i] = s_i$ for all $0 \leq i \leq n$. Then, given an HTN planning problem $\mathcal{P}$, a plan is a solution to $\mathcal{P}$ if there exists an operator sequence applicable in s_0 by decomposing tn_0. The way of decomposing the initial task network and producing an operator sequence is defined in the following sections.

Search space

The basic hierarchical model gives only an idea what hierarchical planning consists of. If we wish to acquire a deeper understanding of HTN planning, we have to look back at its beginnings and representatives. While there are different variants of HTN planning today, at first glance, this variant distinction seems not that obvious and comprehensible. We discover that the structure of the search space in hierarchical planning is not necessarily state based. In fact, there are two structures of search spaces created by hierarchical planners. Let us intuitively describe each one.

The first space structure consists of task networks and task decompositions as evolutions from one task network to another. Given an HTN planning problem $\mathcal{P}$, at the beginning of the search, a task decomposition is imposed on the initial task network tn_0, and the process continues by repeatedly decomposing tasks from a newly created task network until a primitive task network is produced. A linearisation of this primitive task network applicable in the initial state s_0 represents a solution to $\mathcal{P}$.

The second space structure is in essence a subset of the state space. It consists of explicitly described states restricted by task decompositions. As in classical planning, the search begins in s_0 with an empty plan, but instead of searching for a state that will satisfy the goal state, the search is for a state that will accomplish tn_0. In particular, if a task from the task network is compound, the task decomposition continues on the next decomposition level, but in the same state. Otherwise, the task is executed and the search continues into a successor state. The task in the latter case is then added to the plan. When there are no more tasks in the task network to be decomposed, the search is finished. The solution to $\mathcal{P}$ is the plan containing a sequence of totally ordered primitive tasks.

Categorisation of hierarchical planning. The initial task network in the former case is reduced to a primitive task network that constitutes a solution to the planning problem. At each point in the space, the task network can be seen as a partially specified plan until the search reaches the point where the task network is primitive and represents a solution plan. Thus, we employ the term *plan space* to refer to this structure of search space. We refer to HTN planners that search in this plan space as *plan-based HTN planners*, and to the model of HTN planning as *plan-based HTN planning*. For the obvious reasons, we employ the term *state space* to refer to the latter structure of search space. We refer to HTN planners searching in this space as *state-based HTN planners*, and to the model of HTN planning as *state-based HTN planning*.

4.2.1 Plan-based HTN planning

We draw the formalism of plan-based HTN planning upon the work of (Geier and Bercher 2011). We complement Definition 4.2 as follows.

4.5 Definition (Task network). *A task network tn is a triple (T, φ, ψ), where*

- *T is a finite and non-empty set of tasks,*
- *$\varphi : T \to T_n$ labels a task with a task name,*

- *ψ is a formula composed by conjunction, disjunction or negation of the following sets of constraints:*
  - *$\prec \subseteq T \times T$ is a strict partial order on T (irreflexive, transitive, asymmetric),*
  - *$\mapsto \subseteq V \times V \cup V \times C$ is a restriction on bindings of task network variables, and*
  - *$\vdash_{\prec} \subseteq T \times Q \cup Q \times T \cup T \times Q \times T$ is a partial order on tasks and state predicates.*

Since some task name can occur many times in one task network, task labelling enables identifying uniquely many occurrences of that task name. For example, $tn = (\{t_1, t_2, t_3\}, \{(t_1, t'), (t_2, t''), (t_3, t')\}, \emptyset)$ denotes that the task network consists of two tasks associated with task name t' and one task associated with t''.

A task network $tn = (T, \varphi, \psi)$ is *isomorphic* to $tn' = (T', \varphi', \psi')$, denoted as $tn \equiv tn'$, if and only if there exists a bijection $\beta : T \to T'$, such that

- for all $t, t' \in T$ it holds $(t, t') \in \prec$ if and only if $(\beta(t), \beta(t')) \in \prec'$,
- for all $v_1, v_2 \in V$ and $c \in C$ it holds $(v_1, v_2) \in \mapsto$ or $(v_1, c) \in \mapsto$ if and only if there exist $v_1', v_2' \in V$ and $c' \in C$ such that $v_1 = v_1'$, $v_2 = v_2'$ and $(v_1', v_2') \in \mapsto'$ or $v_1 = v_1'$, $c = c'$ and $(v_1', c) \in \mapsto'$,
- for all $t, t' \in T$ and $q \in Q$ it holds $(t, q) \in \vdash_{\prec}$ or $(q, t) \in \vdash_{\prec}$ or $(t, q, t') \in \vdash_{\prec}$ if and only if $(\beta(t), q) \in \vdash'_{\prec}$ or $(q, \beta(t)) \in \vdash'_{\prec}$ or $(\beta(t), q, \beta(t')) \in \vdash'_{\prec}$,

and $\varphi(t) = \varphi'(\beta(t))$.

4.6 Definition (Decomposition). *Let m be a method and $tn_c = (T_c, \varphi_c, \psi_c)$ be a task network. Method m* decomposes *tn_c into a new task network tn_n by replacing task t, denoted as $tn_c \underset{t,m}{\longrightarrow_D} tn_n$, if and only if $t \in T_c$, $\varphi_c(t) = c(m)$, and there exists a task network $tn' = (T', \varphi', \psi')$ such that $tn' \equiv tn(m)$ and $T' \cap T \neq 0$, and*

$$
\begin{aligned}
tn_n :=& ((T_c \setminus \{t\}) \cup T', \varphi_c \cup \varphi', \psi_c \cup \psi' \cup \psi_D) \text{ where} \\
\psi_D :=& \{(t_1, t_2) \in T_c \times T' \mid (t_1, t) \in \prec_c\} \cup \{(t_1, t_2) \in T' \times T_c \mid (t, t_2) \in \prec_c\} \cup \\
& \{(q, t_1) \in Q \times T' \mid (q, t) \in \vdash_{\prec c}\} \cup \{(t_1, q) \in T' \times Q \mid (t, q) \in \vdash_{\prec c}\} \cup \\
& \{(t_1, q, t_2) \in T' \times Q \times T' \mid (t, q, t_2) \in \vdash_{\prec c}\}
\end{aligned}
$$

Given an HTN planning problem $\mathcal{P}$, $tn_c \to_D^* tn_n$ indicates that tn_n results from tn_c by an arbitrary number of decompositions using methods from M.

4.7 Definition (Executable task network). *Given an HTN planning problem $\mathcal{P}$, $tn = (T, \varphi, \psi)$ is* executable *in state s, if and only if it is primitive and there exists linearisation of its tasks $t_1, \ldots, t_n$ that is compatible with ψ and the corresponding sequence fo operators $\varphi(t_1), \ldots, \varphi(t_n)$ is executable in s.*

4.8 Definition (Solution). *Let $\mathcal{P}$ be an HTN planning problem. A task network tn_s is a solution to $\mathcal{P}$, if and only if tn_s is executable in s_0, and $tn_0 \rightarrow_D^* tn_s$ for tn_s being a solution to $\mathcal{P}$.*

Intuitively, a problem space is a directed graph in which task networks are vertices, and a decomposition of one task network into another task network by some method is an outgoing edge, under the condition that the initial task network belongs to the graph (similarly to the definition of the decomposition problem space in (Alford et al. 2012)).

4.9 Definition (Plan space). *Given a (plan-based) HTN planning problem $\mathcal{P}$, a* plan space *PG is a directed graph $(\mathcal{V}, \mathcal{E})$ such that $tn_0 \in \mathcal{V}$, and for each $tn \rightarrow_D tn'$: $tn, tn' \in \mathcal{V}$ and $(tn, tn') \in \mathcal{E}$.*

4.2.2 State-based HTN planning

We complement Definition 4.2 and 4.3 of the basic hierarchical planning model as follows.

4.10 Definition (Task network). *A task network tn is a pair $(T, \prec)$, where T is a finite set of tasks, and $\prec$ is a strict partial order on T (irreflexive, transitive, asymmetric).*

A task network tn in state-based HTN planning is less expressive than the one in plan-based HTN planning. Here, tn does not allow multiple occurrences of a same task in the partial ordering of tasks.

4.11 Definition (Method). *A method m is a triple $(c(m), pre(m), tn(m))$, where $c(m)$ is a compound task, $pre(m) \in 2^Q$ is a precondition, and $tn(m)$ is a task network. The subsets $pre^+(m)$ and $pre^-(m)$ denote positive and negative precondition of m, respectively.*

A method m is applicable in state s, if and only if $pre^+(m) \subseteq s$ and $pre^-(m) \cap s = \emptyset$. Applying m to s results in a new task network.

4.12 Definition (Decomposition). *Let m be an applicable method in s and $tn_c = (T_c, \prec_c)$ be a task network. Method m decomposes tn_c into a new task network tn_n by replacing task t, written $tn_c \underset{s,t,m}{\longrightarrow_D} tn_n$, if and only if $t \in T_c$, $t = c(m)$ and*

$$tn_n := ((T_c \setminus \{t\}) \cup T_m, \prec_c \cup \prec_m \cup \prec_D) \text{ where}$$
$$\prec_D := \{(t_1, t_2) \in T_c \times T_m \mid (t_1, t) \in \prec_c\} \cup \{(t_1, t_2) \in T_m \times T_c \mid (t, t_2) \in \prec_c\}$$

4.13 Definition (Solution). *Let $\mathcal{P}$ be an HTN planning problem. The sequence $o_1, \ldots, o_n$ is a solution to $\mathcal{P}$, if and only if there exists a task $t \in T_0$, where $tn_0 = \langle T_o, \prec_0 \rangle$, such that $(t, t') \in \prec_0$ for all $t' \in T_0$ and*

- *t is primitive and applicable in s_0 and the sequence $o_2, \ldots, o_n$ is a solution to $\mathcal{P}$ in which the task network is $tn_0 \setminus \{o_1\}$ and the state is $s_0[o_1]$; or*
- *t is compound and there is a task decomposition in s_0 such that the sequence $o_1, \ldots, o_n$ is a solution to $\mathcal{P}$ in which $tn_0 \rightarrow_D tn'$.*

We consider a state space as a directed graph in which a state is a vertex, and a task decomposition maps to the same state where the corresponding method is applicable, and an operator application leads to a successor state. A slightly different approach is presented in (Alford et al. 2012), where the space is a directed graph in which pairs of state and task network are vertices, and a progression from one pair to another is an outgoing edge.

4.14 Definition (State space). *Given a (state-based) HTN planning problem $\mathcal{P}$, a* state space *SG is a directed graph $(\mathcal{V}, \mathcal{E})$ such that $s_0 \in \mathcal{V}$, and there is a state s_i and $t_k \in tn$ such that*

- *if t_k is primitive, then $s_i[t_k] = s_{i+1}$ such that $k = i+1$, $s_i, s_{i+1} \in \mathcal{V}$ and $(s_i, s_{i+1}) \in \mathcal{E}$; or*
- *if t_k is compound, then $tn \rightarrow_D tn'$ is a self-transition such that $s_i \in \mathcal{V}$ and $(s_i, s_i) \in \mathcal{E}$.*

4.3 Concepts

There is indeed a large body of literature on hierarchical planning, reporting however vague and ambiguous information on planning concepts and planners especially in the beginning of the development of the field of HTN planning. The literature accommodates pitfalls related to the clarity of exploited ideas and concepts, and how they are adapted for the purpose of HTN planning. We can notice a slight improvement in clarifications at the time of the appearance of new HTN planners and the attempts at formalisation.

We clarify this condition by designing and developing a conceptual model related to the search space of hierarchical planning. Contrary to the hierarchical planning model, the conceptual model is less formal and describes specific concepts derived from empirical observation. While the descriptions of concepts we provide are basic and general, they characterise different HTN planners and cover most of their important features. The concepts are placed within a logical and sequential design as much as possible. The key concept of the model is the search space to which other concepts are related and interconnected in various ways. Considering the approach of conceptual modelling presented in Section 3.1, we construct the UML model specification shown in Figure 4.1.

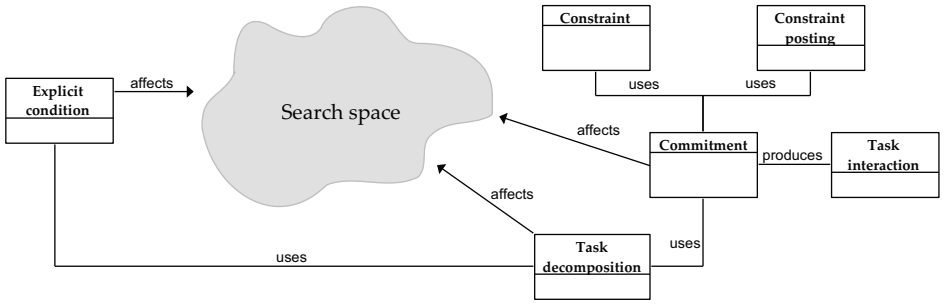


Figure 4.1: Concepts and their relationships that affect the search space of HTN planners specified in UML.

4.3.1 Task decomposition

Given a task network tn, a *task decomposition* chooses a task t from tn and, if t is primitive and applicable to the current state s, the task decomposition applies t to s. Otherwise, the decomposition strategy analyses all the methods that contain t as a part of their definition. Assuming that a set of methods is found, the task decomposition makes a non-deterministic choice of a method m, and replaces t with the task network associated with m. Finally, the task decomposition checks the newly composed task network against any constraint-related violation and modifies it, if necessary.

We divide task decompositions into three styles based on the representation of task networks in terms of task ordering, and the way of forming new task networks during decomposition. The first one is *totally ordered task decomposition* (TOTD). It follows the assumption of total order on task networks so as when a task is decomposed, the new task network is created in such a way that newly added tasks are totally ordered among each other and with respect to the tasks of the existing task network. Sometimes we refer to the HTN planning that uses this style as totally ordered HTN planning. The second style is *unordered task decomposition* (UTD) that relaxes the requirement of totally ordered task networks. That is, tasks can be totally ordered or unordered with respect to each other (but no tasks in parallel are allowed). When a task is decomposed, new task networks are created in such a way that newly added tasks are interleaved with the tasks of the existing task network until all permissible permutations are exhausted. Here as well, we refer to the HTN planning that embodies this style as unordered HTN planning. The last style is *partially ordered task decomposition* (POTD) that allows the existence of a partial order on tasks. When a task is decomposed, the tasks in the newly created network can be ordered in parallel whenever possible (with respect to the constraints). We

refer to the HTN planning that uses this style as partially ordered HTN planning.

4.3.2 Constraints

Looking back at Definition 4.2, we observe that a task network relies upon the constraints provided in the problem representation. Also, constraints can be added during planning in order to resolve inconsistencies. Hierarchical planners deal with several types of constraints and most of them can be interpreted as in (Stefik 1981). Namely, there are three interpretations. First, we meet a constraint that implies commitments about partial descriptions of state objects. Another type of constraint refines variable bindings if a certain variable binding does not satisfy some condition. Last, there is a constraint that expresses the relations between variables in different parts of a task network.

Commitment strategy

As with the planners in classical planning, hierarchical planning techniques also need to make two decisions on constraints. The first one is on constraints for binding variables, while the second decision is on constraints for ordering tasks in a task network. We extract two main approaches for when and how to make these decisions. The first approach manages constraints in compliance with the *least-commitment strategy* so that task ordering and variable bindings are deferred until a decision is forced (Weld 1994). The second approach handles constraints according to the *early-commitment strategy* so that variables are bound and operators in the plan are totally ordered at each step of the planning process. Planners employing the latter strategy greatly benefit from the possibility of adopting forward chaining in which chaining of operators is achieved by imposing a total order over the plan. The total ordering ensures that neither the current operator to be added to the plan can interfere with some earlier operator's preconditions or effects, nor a later operator can interfere with current task's preconditions or effects.

Task interaction

An inevitable consequence of a commitment strategy is the interaction among tasks in a given task network. We define an *interaction* as the connection between two tasks (or parts) of a task network in which these tasks (or parts) have a certain effect on each other. Based on this effect, we divide interactions into two categories. The first category introduces conflicts among different parts of a task network that threaten its correctness. We refer to this category as *harmful interactions* (also threats or flaws). While HTN planners differentiate various harmful interactions, there are rather intuitive descriptions provided for each. We abstract definitions of several

harmful interactions in the following list.

- *Deleted-condition interaction* happens when a primitive task in one part of a task network deletes an expression that is a precondition to a primitive task in another part of that task network.

- *Double-cross interaction* appears when an effect of each of two conjunctive primitive tasks deletes a precondition for the other. That is, an effect of the first task deletes a precondition of the second primitive task, and an effect of the second task deletes a precondition of the first task.

- *Resource interaction* occurs in two situations, and it is subdivided accordingly. A *resource-resource interaction* is similar to the deleted-condition interaction, while a *resource-argument interaction* occurs when a resource (see Definition 4.15) in one part of a task network is used as an argument in another part of that task network.

The second category involves situations when one part of a task network can make use of information associated with another part in the same task network. We refer to this category as *helpful interactions*. The detection of these interactions implies the possibility for a planner to generate better-quality task networks and solutions. That is, some tasks can be merged together eliminating task redundancy and potentially optimising the cost of the solution (Foulser et al. 1992). We provide descriptions of several helpful interactions in the following list.

- *Placeholder replacement* appears when a real value already exists for a particular formal object. We already know that HTN planning allows tasks with variables to be inserted into a task network. If there is no specific value to be chosen for a particular variable choice, a so-called *formal object* is created to bind the variable (Sacerdoti 1975b). The formal object is simply a placeholder for some entity unspecified at that point.

- *Phantomisation* emerges when some goal is already true at the point in a task network where it occurs. In the descriptions of some HTN planners, the term 'goal' is interchangeably used with the term 'precondition' – if some precondition is not satisfied, it is inserted as a goal to be achieved (as in classical planning).

- *Disjunct optimisation* happens in disjunctive goals when one disjunctive goal is "superior to the others by the nature of its interaction" with the other tasks in a task network (Sacerdoti 1975b).

Constraint management

HTN planners do not provide a general approach for handling interactions, thus each of the above interactions has its own resolution method. The underlying process of solving task interactions is *constraint posting* (also known as conflict resolution (Yang 1992) or critics (Tate 1976, Wilkins 1988)), which manipulates various types of constraints in a task network. The process is based on well-known operations on constraints generally described elsewhere, *e.g.,* (Stefik 1981). We briefly explain the main operations in the context of hierarchical planning.

The most basic operation is *constraint satisfaction* which happens when a hierarchical planner searches for a variable binding that satisfies the given constraints, and guarantees the consistency of, for instance, a set of ordering constraints over a task network. *Constraint propagation* enables adding or retracting constraints to and from a task network. Variable constraints in one part of a task network can be propagated based on variable constraints in another part of that task network. With respect to ordering constraints, propagation is used when a linking process is performed. When some task interferes with another task, the *linking process* records a causal link – a three-element structure of two pointers to tasks t_e and t_p, and a predicate q which is both an effect of t_e and a precondition of t_p. This linking process practically achieves phantomisation. That is, phantomisation of a task t with an effect e is accomplished by treating e as achieved, and finding an existing task t' in the task network that achieves the same effect e. If task t' is found, a constraint (t', e, t) is added to the task network to record the causal relation.

The last operation is different in that it does not happen during planning. *Constraint formulation* can be taken into account when modelling HTN domain knowledge, especially when the domain author is aware in advance of some possible impasse situations. By posting constraints as control information into the domain knowledge, the planner can gain on efficiency by refining the search space (Erol et al. 1994b, Nareyek et al. 2005). Moreover, in some HTN planners, the phantomisation of a task is achieved by an explicit encoding in the domain knowledge. The planners handle the phantomisation of a rather recursive task by taking into account an alternative method that encodes the *base case* explicitly through a 'do-nothing' operation. There is also a possibility for such planners to infer these situations automatically, which we cover in Section 5.2.

4.3.3 Explicit conditions

Hierarchical planners essentially depend on the quality of domain knowledge so as to restrict and guide the search for a solution. The domain author is undoubtedly the one who has the responsibility of giving the guidance information. One way to

represent such information is by using explicit conditions. We describe conditions found in HTN planners in the following.

- *Supervised condition* is accomplished within a compound task. The condition may be satisfied either by an intentional insertion of a relevant effect earlier in the processing of a task network, or by an explicit introduction of a primitive task that will achieve the required effect. Only this condition should allow further decompositions to be made.

- *External condition* must be accomplished at the required task, but under the assumption that it is satisfied by some other task from the same task network.

- *Filter condition* decides on task relevance to a particular situation. In the case of method relevance to a certain task decomposition, this condition reduces the branching factor by eliminating inappropriate methods.

- *Query condition* accomplishes queries about variable bindings or restrictions at some required point in a task network.

- *Compute condition* requires satisfaction by information coming only from external systems, such as a database.

- *Achieve condition* allows expressing goals that can be achieved by any means available to a planner.

4.3.4 Overview of planners

We here provide a summary of task decomposition, constraints and constraint-based techniques, and explicit conditions as specified in each hierarchical planner. We summarise in a tabular form for which we use the following notation. We use a grey shade to visually separate planners in plan based and state based (grey-shaded columns and rows signify state-based HTN planners). If a cell contains '✓', then a planner supports or defines the respective element. A cell with '✗' indicates no support or definition of an element, while '-' denotes that the planner does not need to support or handle the respective element. If a cell is empty, then it means that the information was not available from the public literature.

Table 4.1 demonstrates the concept of a task decomposition as realised in hierarchical planners. Since the task decomposition depends on the representation of tasks and task networks, we provide insights into how primitive and compound tasks are represented (column **Mechanism for primitive and compound tasks**), and what a task network consist of (column **Task network**). In the column **Task**

decomposition, we show 1) how the decomposition of a 'compound' task is accomplished, and how a 'primitive' task is applied (column **Process**); 2) the style of a task decomposition (column **Type**); and 4) whether a task network is checked against any constraint violation during task decomposition (column **Constraint check**).

Most plan-based HTN planners perform a task decomposition in a slightly different way than the general process we described in Section 4.3.1. The main reason lies in the approach that these planners use to represent tasks. In fact, with the exception of UMCP, the rest of the planners support only a single structure to encode both primitive and compound tasks. Although it is not always clear what is the purpose of the respective structure or how exactly the task decomposition is accomplished, we try to make high-level statements on the main idea behind the decomposition at each planner. For instance, the statement "D: code" denotes that the decomposition of a 'compound' task in NOAH is accomplished by an evaluation of the respective node's code, but also the application of a 'primitive' task is done by evaluating that code ("A: code").

State-based HTN planners in essence follow the task decomposition from Section 4.3.1, and indeed distinguish between primitive and compound tasks. In both planners, SHOP2 and SIADEX, the set of methods can be seen as an if-then-else representation – the planners select the first method whose if-statement (preconditions) holds in the current state. Given a compound task, a task decomposition evaluates the preconditions of task's associated methods, and chooses the first method applicable in the current state to expand the existing task network.

We accentuate two observations. In the case of both planners, the choice of which method to use for the decomposition is controlled, that is, the first method from the if-then-else representation that is applicable in the state is chosen. Second, SHOP2 uses the unordered task decomposition (its predecessor SHOP employs TOTD), while SIADEX follows the partially ordered task decomposition. Consequently, SHOP2 does not need to check the task network for corrections during task decomposition, but SIADEX must verify that no (ordering) constraints are violated in the newly created task network.

The constraint-related concepts, namely the commitment strategy and constraint management in the case of task interactions are shown in the upper part of Table 4.2. Plan-based HTN planners take advantage of the least-commitment strategy, however, we point out two deficiencies. First, except UMCP, which supports additional commitment strategies (Tsuneto et al. 1996), the rest of the planners take a rigid approach of incorporating the commitment strategy into the problem-solving mechanism. Second, only few planners backtrack on poor decisions, thus not implementing the concept of the least-commitment strategy completely. On the other hand, state-based HTN planners employ the early-commitment strategy.

Table 4.1: Task decomposition in state-of-the-art hierarchical planners ("D" and "A" stand for "decomposition" and "application", respectively).

	Hierarchical representation		Task decomposition		
	Mechanism for primitive and compound tasks	**Task network**	**Process**	**Type**	**Constraint check**
NOAH	single for both: node (code)	network of nodes	D: code A: code	POTD	✓
Nonlin	single for both: node (schema)	network of nodes	D: schema A: schema	POTD	✓
SIPE-2	single for both: node (operator)	network of nodes	D: operator A: - (effect deduction)	POTD	✓
O-Plan2	single for both: schema	network of schemas	D: schema A: schema	POTD	✓
UMCP	separate for each	network of primitive and compound tasks	D: compound task A: primitive task	POTD	✓
SHOP2	separate for each	network of primitive and compound tasks	D: compound task A: primitive task	UTD	-
SIADEX	separate for each	network of primitive and compound tasks	D: compound task A: primitive task	POTD	✓

If some task fails, both planners backtrack on other alternatives according to a list of variable bindings for the task precondition, or maybe to some criterion specified in the definition of the task. In addition, SIADEX supports cutting of backtracking points (as performed in Prolog (Bratko 2001)).

The lower part of Table 4.2 summarises and classifies resolution methods with respect to the task interaction they solve. NOAH and SIPE-2 need to handle the largest set of interactions, while Nonlin and UMCP handle only one harmful and one helpful interaction. As for state-based HTN planners, if we consider their underlying early-commitment strategy, we could conclude that these planners avoid task interactions altogether. However, this statement would not be entirely correct. The lower part of Table 4.2 shows that in SHOP2, for instance, a deleted-condition interaction may arise and this is due to the process of interleaving tasks. The planner is able to solve this situation under a rather restricting assumption, that is, it requires a specification of 'protection' conditions in the effects of operators. A protection request enforces the planner from deleting conditions, and a protection cancellation allows the planner to delete these conditions. In some cases, the planner can deal with deleted-condition interaction using domain axioms. SIADEX needs a more powerful mechanism to accomplish planning and handle interactions that may arise in partially ordered task networks. The planner uses a causal structure of tasks and task networks. Constraint satisfaction checks the consistency of task networks (and the solution) based on that causal structure, and constraint propagation is used to post constraints, if necessary.

Table 4.3 summarises and classifies explicit conditions that hierarchical planners employ. We observe that Nonlin initiated the idea of explicit types, supporting four types of conditions. O-Plan2 supports the largest set of conditions, where they play some "special role" in the planner's planning process (Tate, Drabble and Kirby 1994). In UMCP, conditions are represented as state constraints. In addition to explicitly typing them into the domain knowledge, the planner is extended to reason about implicit external conditions by examining the domain knowledge (Tsuneto et al. 1998).

The whole reasoning power of SHOP2 and SIADEX is encapsulated in the preconditions of both primitive and compound tasks, thus they do not require other explicit domain knowledge. In the scope of preconditions, however, SHOP2 enables various types of computations, such as invocations of external knowledge resources by using the `Call` condition. SIADEX also supports complex computations by incorporating complete (Python-based (*Python* 2014)) procedures in the domain. External conditions are modelled in a similar fashion.

Table 4.2: Constraint-related concepts in state-of-the-art hierarchical planners.

Commitment	**NOAH**	**Nonlin**	**SIPE-2**	**O-Plan2**	**UMCP**	**SHOP2**	**SIADEX**
Strategy	least	least	least	least	least (+other)	early	early
Backtracking	no	yes	partially	yes	yes	yes	yes
Interaction							
Deleted-condition	Resolve conflicts	Linking process	Solving harm. inter.	TOME/GOST managers	Resolution method	Protection cond., domain axiom	Constraint propagation
Double-cross	Resolve double cross	-	Solving harm. inter.	-	-	×	×
Resource	×	×	Resource conflicts	Resource util. managers	×	×	×
Placeholder replacement	Use existing objects					×	×
Phantomisation	Eliminate redundant preconditions	Linking process	Goal phantom.	Question answering	Domain method	×	×
Disjunct optimisation	Optimise disjuncts				-	×	×

Table 4.3: Condition types in state-of-the-art hierarchical planners (adapted from (Tate, Drabble and Dalton 1994))

Condition	**NOAH**	**Nonlin**	**SIPE-2**	**O-Plan2**	**UMCP**	**SHOP2**	**SIADEX**
Supervised	prec.	supervised	protect-until	supervised	goal task	×	×
External	×	unsupervised	external-cond.	unsupervised	constr./external	call	arbitrary code
Filter	×	use-when	prec.	only_use_if	filter	prec.	prec.
Query	×	×	×	only_use_for_query	×	×	×
Compute	×	×	×	compute	×	call	function
Achieve	×	×	×	achieve at N	×		
	goal	goal	goal	achieve at N after <time point>	goal task	×	×

4.4 Properties

We go further and consider another perspective of HTN planning and planners. We cover properties of HTN planners related mainly to domain knowledge, expressiveness, soundness, completeness, fault tolerance, performance, and applicability. Our motivation for this step essentially lies in some of the implicit assumptions made about HTN planners, that is, claims and beliefs accepted for granted and without evidence. Some of these refer to the "sophistication" of domain knowledge provided to HTN planners, the expressive power of HTN planning between theory and practice, HTN planners being fast and scalable, and HTN planning being very suitable for and most applied to real-world problems.

We develop an *analytical model* to directs us on where to look and what kind of properties to look for. The model consist of five main elements, namely domain authoring, expressiveness, competence, computation, and applicability. Each element and the motivation for its inclusion in the model are described in the following sections. Whenever possible, we provide formal definitions that may be related to the models presented in Section 4.2. We then collect a body of studies on HTN planning and planners, and apply exploratory research to examine diversity and similarity of the planners within their category and between categories, and comparative research to make sense of a range of cases. We believe that, in this way, we can rectify the statements about HTN planning in a neutral and evidence-oriented way.

4.4.1 Domain authoring

We define *domain authoring* as the formulation of domain knowledge as performed by a domain author. What we are really interested in, in this process, is the relative effort needed to formulate domain knowledge for an HTN planner. The community, however, has not yet found a way or measures to provide an objective answer to this type of question. The ambiguity and difficulty to define an answer come directly from the capabilities and experience of the domain author with respect to the understanding of the underlying planner and the expertise for the respective domain (Long and Fox 2003).

The domain knowledge plays a crucial role in HTN planning. Therefore, we still want to give a flavour of the effort needed to provide domain knowledge for each planner. We take a model of the well-known and overused domain of blocks world (Chenoweth 1991) as described for each planner, and inspect each model from two viewpoints. We first take a single and the same task of each domain model and analyse closely what needs to be encoded. Second, we give a broader view of each domain model by quantifying its content with respect to knowledge symbols, keyword symbols, and domain elements.

4.4.2 Expressiveness

We tackle expressiveness from two perspectives. The first one refers to the formal properties of expressiveness of the HTN planning language. In order to completely determine what the language can express, we need formal semantics for the language. Fortunately, this issue has been a subject of interest for some time, resulting in a number of studies on expressiveness of HTN planning (Erol et al. 1994a, Kambhampati 1995, Nau et al. 1998, Wilkins and Desjardins 2001, Lekavý and Návrat 2007, Erol 1996). We analyse the expressiveness of HTN planning language from a model-theoretic, operational and computational aspect based on the results provided in (Erol et al. 1996). In each aspect, the expressiveness of HTN planning is compared to the one of STRIPS planning (see Section 2.1).

The second perspective refers to the practical expressive power of HTN planners. We could determine the practical expressiveness of the planners' language by the assessment of the breadth of what the language can represent and communicate. The breadth may include the language's formal system, the support for preferences, time, *etc.*. Unfortunately, there is no common planning language for HTN planners. The idea of standardising a planning language is introduced with PDDL in 1998 for the purpose of the International Planning Competition, rather late with respect to the history of, above all, plan-based HTN planners. Although in the first version of PDDL there was an attempt to formalise a common syntax compatible to HTN planners, the idea was discarded with version 2.1 of PDDL due to the immense differences between planners (Fox and Long 2003).

We can still provide some insights about what HTN planners can express in practice by exploring the expressiveness of each planner's language separately. For this purpose, we use three categories of properties. The first category encompasses the system of first-order logic, particularly the support for a set of logical connectives: conjunction ($\wedge$), disjunction ($\vee$), implication ($\Longrightarrow$), and negation ($\neg$), and the support for universal ($\forall$) and existential ($\exists$) quantifiers. A logical connector or quantifier can be applied on $pre(o)$ and $eff(o)$, where o is an operator, on the formula ψ of a task network tn, and on $pre(m)$, where m is a method.

The second category encompasses the quality constraints that can be expressed in some languages, particularly the support for typing, extended goals, and preferences. Given an HTN planning problem $\mathcal{P}$, we define each as follows.

- *Typing* enables expressing types of objects in a type hierarchy (similar to typing in PDDL). Each $v \in V$ may have a type $\mathbf{t} \in \mathbf{T}$, where $\mathbf{T}$ is a set of types.[2] The type hierarchy is built by a sub-typing relation $st : \mathbf{T} \times \mathbf{T}$, which is reflexive

[2]We use letters in bold for sets defined in encoding languages to differentiate from the sets of the HTN planning language.

and transitive.

- HTN planning assumes an initial task network tn_0 to be accomplished as an objective for $\mathcal{P}$. In its simplest form, tn_0 does not allow to specify conditions to be satisfied in some intermediate state during or in the final state of the execution of the solution to $\mathcal{P}$. *Extended goals* enable us to express a planning objective in a way that its satisfaction could be on a part, on the whole trajectory of the solution, or in the final state. In classical planning, this is usually achieved through the use of temporal modal operators.
- A *preference* is a condition on the solution trajectory that some user would prefer satisfied rather than not satisfied, but would accept if the condition might not be satisfied (Gerevini and Long 2006).

The third category encompasses resource and time constraints.

4.15 Definition (Resource). *Given an HTN planning problem $\mathcal{P}$, a* resource *r is an object of limited capacity $\bar{r}$ for use by a task t within $\mathcal{P}$.*

The capacity $\bar{r}$ can be a categorical value, such as free (for use) and used, or a numerical value. If $c_i(r)$ denotes the current capacity of a resource r, then $c_0(r) = \bar{r}$ is the initial capacity. We use $c_t(r)$ to denote the consumption of r by a task t. We use $t_1 \parallel t_2$ to denote that tasks t_1 and t_2 are in parallel, and $c_{t_1}(r) \parallel c_{t_2}(r)$ for the consumption of resource r by tasks t_1 and t_2 at the same time. Thus, $c_{t_1}(r) \parallel c_{t_2}(r)$ is possible iff $t_1 \parallel t_2$.

4.16 Definition (Types of resources). *A resource is reusable if it can be used more than once. Let t be a task and r be a resource whose $\bar{r}$ has a categorical value. The resource r is reusable iff $c_i(r) = c_0(r)$ immediately after $c_t(r)$.*

A shared reusable resource can be shared among several tasks at the same time. Let t_1 and t_2 be tasks and r be a resource. The resource r is shared reusable iff r is reusable and $c_{t_1}(r) \parallel c_{t_2}(r)$.

An exclusively reusable resource cannot be used by two tasks in parallel. Let t_1 be a task and r a resource. The resource r is exclusively reusable iff r is reusable and $c_{t_1}(r)$ such that $c_{t_1}(r) \nparallel c_{t_i}(r)$, where $i > 1$.

A resource is consumable if it is usable only a limited number of times. Let t be a task and r be a resource whose $\bar{r}$ has a numerical value. The resource r is consumable iff $c_i(r) = c_{i-1}(r) - c_t(r)$ immediately after $c_t(r)$. A consumable resource can be replenished or not.

If the resource cannot be restored after the use of the set amount, it is called disposable consumable resource. Let t be a task and r a resource. The resource r is disposable consumable iff r is consumable, and there exists i such that $c_i(r) = 0$ and there is no $i + k$, $k \in \mathbb{N}$ such that $c_{i+k}(r) > 0$.

If the resource amount can be topped up, it is called renewable consumable resource. Let t be a task and r a resource. The resource r is renewable consumable iff r is consumable, $c_i(r) = c_{i-1}(r) - c_t(r)$ and there exists $o \in O$ and $k \in \mathbb{N}$ such that $c_{i+k}(r) = c_i(r) + rep$, where $rep \in eff(o)$.

Finally, we define time as usually considered, that is, a consumable resource that cannot be reproduced. We are interested in how HTN planners represent and handle temporal information.

4.4.3 Competence

We use the term *competence* to encompass a category of functional and formal properties that relate to specific abilities of HTN planners. We begin with properties of the functional design of hierarchical planners.

Domain dependence defines the ability of a planning technique to solve planning problems in different domains (Nau 2007). This is the issue of *domain-specific planning*, which is designed to solve problems only in a particular domain, versus *domain-configurable planning*, which solves planning problems in any domain given specific knowledge for every domain, versus *domain-independent planning*, which solves planning problems in any domain without specific demands. Given that HTN planning can solve problems in various domains, and it requires specific-domain knowledge provided in the set of methods M, *HTN planning is a domain-configurable planning technique*. This implies a design and implementation of HTN planners that include general problem-solving mechanisms. A problem-solving mechanism takes in a given $\mathcal{P}$ and computes a solution. It makes use of various algorithms and backtracking mechanisms, heuristics, specific control knowledge, and constraint management. Thus, several options arise for the design of a planner's mechanism:

- *Algorithm* represents the search procedure incorporated in the problem-solving mechanism. The mechanism can employ one or more kinds of search strategies, such as depth-first search (DFS), breadth-first search (BrFS), iterative deepening search (IDS), best-first search (BFS), or other heuristic search (HS) approaches. The algorithm incorporates the process of task decomposition (see Section 4.3.1), and may traverse the data structure of $\mathcal{P}$ with or

without *backtracking* points. Recall from Section 4.3.2 that in HTN planning there are three types of backtracking or decision points: which task to deal with next, which method to use for some task, and which value to bind to a variable.

- *Heuristics* are functions that help the problem-solving mechanism to guide and speed up the search for a solution. In some HTN planners, heuristics may trade completeness for speed.
- *Domain-specific control* is represented by the set of methods M. The problem-solving mechanism evaluates the preconditions of methods in M to guide the planning process.
- *Interactive control* involves user's decisions during planning. A user may guide the problem-solving mechanism by choosing values to bind to variables in V, imposing ordering constraints on a set of tasks T, and decomposing the current task network tn.
- *Constraint management* (CM) deals with constraints that are part of task networks of methods in M, and those that can be added during planning. The problem-solving mechanism makes use of the constraint-related operations discussed in Section 4.3.2.

The following formal properties show when an HTN problem-solving mechanism (or, equally, an HTN planner) is sound and complete, and when the solution that the mechanism generates is flexible.

Solution flexibility defines the ordering of operators in the solution to a planning problem.

4.17 Definition (Flexibility). *Let $\mathcal{P}$ be an HTN planning problem. The solution to $\mathcal{P}$ is* flexible *if it is partially ordered.*

4.18 Definition (Soundness). *Let $\mathcal{P}$ be an HTN planning problem. An HTN planner is* sound *if every plan it gives is a correct solution to $\mathcal{P}$.*

4.19 Definition (Completeness). *Let $\mathcal{P}$ be an HTN planning problem. An HTN planner is* complete *if it always finds a solution to $\mathcal{P}$ when such a solution exists.*

Problem-solving mechanisms perform off-line planning with the closed-world assumption: the state can only be changed by the execution of operators selected by the planner. However, this is not the case in real-world environments, which are of a complex and dynamic nature and include other agents executing their own independent actions. During the execution of a plan, some unexpected event may occur

that invalidates the solution being executed. If an event represents a state transition, then, from a planner perspective, the environment changes its state as a result of two event classes: plan operators and fault occurrences. The planner itself is responsible for the selection of plan operators. Otherwise, the planner sees an unexpected or unwanted state transition in the environment as a *fault*. The augmentation of an HTN planner with the ability to handle faults in a well-defined way at execution time makes the planner *fault tolerant*. In order to do so, it is a prerequisite to specify the set of faults that an HTN planner can handle. In the event of a fault at execution time, the planner must monitor and recognise the fault, deduce the parts of the solution that are affected by the fault, and repair the existing and affected part of the plan, or re-plan for a new solution. Thus, the planner must ensure that there is a valid plan that accomplishes the initial task network. In the following, we define the notion of fault and fault tolerance formally.

4.20 Definition (Correct execution). *Let $\mathcal{P}$ be an HTN planning problem, $\pi = \{o_1, \ldots, o_n\}$ be the solution to $\mathcal{P}$, where $s_0[\pi] = s_n$. Let π_e be the partially applied (or executed) part of π, and π_r the remaining part still to be executed. The execution of π is in a correct state s iff $s[\pi_r] = s_n$.*

4.21 Definition (Fault). *A fault f is a state transition $s[f] = s'$ such that $f \notin O$ and $s'[\pi_r] \neq s_n$. The fault set is denoted as F.*

4.22 Definition (Fault tolerance). *Let $\mathcal{P}$ be an HTN planning problem and π be a solution to $\mathcal{P}$. An HTN planner is said to* tolerate faults from a fault set F *during the execution of π iff for each $f \in F$, there exists a sequence of operators π_f such that π_f is a solution to $\mathcal{P}$.*

4.4.4 Computation

Computation can be analysed from two perspectives too. The first one refers to the theoretical computational boundaries of HTN planning. For details on this perspective, we refer to the results presented in (Erol et al. 1996), where the plan existence for an HTNplanning problem is analysed under various assumptions.

We deal with the second perspective, which refers to the computational performance of HTN planners. We are interested in the runtime and scalability results of each HTN planner. We say that a planner is *scalable* if it is capable to cope and *acceptably* perform under a varying size of planning problems. Anything but easy is to define dimensions that could measure the size of a problem, nevertheless, scalability is highly desirable in practical settings with an increasing and large number of facts about the state, a large number of users, and a large number of tasks. We are also interested in how well planners scale relative to one another assuming increasingly difficult problems. As for *runtime*, we are interested in pairwise comparisons

between HTN planners with respect to the amount of time they spend on the same sets of problems.

4.4.5 Applicability

Applicability is the last element of the framework and concerns the use of planners in actual applications. It appears to be orthogonal to previous elements, but we have two reasons for its inclusion. First, we strongly believe that the ultimate objective of research on automated planning must be exploitation of planners in a variety of real applications. Oil spills (Agosta 1996), spacecraft assembly (Aarup et al. 1994), microwave manufacturing (Smith et al. 1997), smart spaces (Kaldeli et al. 2012), and Web service composition (Kuter et al. 2005) are a few prominent examples. Second, HTN planning is promoted as the most applied planning technique for real-world problems (Nau et al. 2005), mostly referring to the applications of SHOP2. Thus, we want to see whether and how HTN planning contributes towards the aforementioned objective, and what is the applicability of state-of-the-art HTN planners.

4.4.6 Overview of planners

We check selected literature for the properties of each element of the analytical model. In one case, we provide theoretical and practical interpretations of the respective element. Where possible, we also show comparison of HTN planners. In some cases, we aggregate the data on planners in tabular form. We use the following common notation for all tables. The '✓' denotes that a planner supports the respective property, the '✗' indicates that a planner does not support a particular property, and an empty cell denotes that it is unknown from the literature whether the planner is able to deal with a given element. There are rated properties, where the rate ranges from '★', denoting limited support for the given property, to '★★★', indicating extended support.

Domain authoring

The first aspect of domain authoring deals with the encoding of the same task in the domain model of each planner. We take the description of the 'put-on' task provided to each HTN planner, and analyse closely its meaning. While here be present a brief summary, we refer for a deeper discussion to (Georgievski and Aiello 2015a). The task descriptions for UMCP, SHOP2 and SIADEX differ only in the notation, but specify almost the same meaning. All operators contain simple applicability preconditions and effects. Beside the representational simplicity, the power of these operators is, however, much weaker than the tasks of the rest of state-of-the-art hierarchical planners. The operators cannot handle situations where some

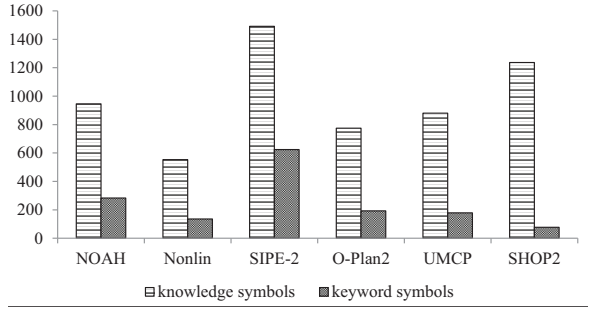
block is above another one or when a block is not clear. The approach to achieve fairly equal functionality would be to include methods that describe all possible situations.

The second aspect of domain authoring gives us insights into encoding domain models by measuring and comparing the sizes of tasks and domain models for each planner, an approach inspired by the one used in (Shivashankar et al. 2012). Although we use domain models for the same domain, we do not assume that the models have the same level of expressiveness. The idea is to establish a relation between the size of a domain and the effort needed to encode that domain. When domain models would have the same level of expressiveness, a smaller domain size would mean that the domain model is easier to encode as compared to the one with a larger size. Looking at the number of symbols at the domain level, as shown in Figure 4.2a, SIPE-2 has the largest domain model, however, almost half of it belongs to keyword symbols. On the other hand, SHOP2 has slightly smaller domain model than SIPE-2, but the number of keyword symbols is negligible, which means that the rest of the symbols represent the actual domain knowledge. In addition, in UMCP and SHOP2, the knowledge is partitioned in a larger number of tasks as compared to the rest of HTN planners. SHOP2 uses knowledge structured in 13 primitive and compound tasks in total, and 6 axioms, while O-Plan2, for example, uses three tasks in total. There are four main reasons for these observations:

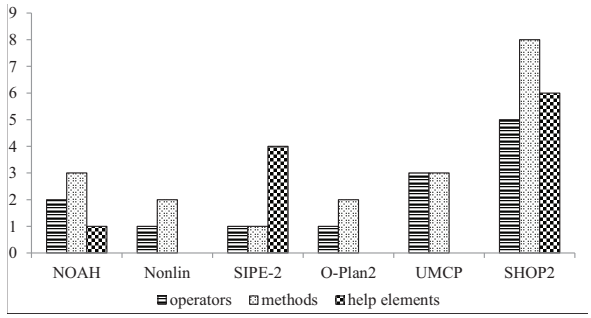
- In SHOP2, a predicate q can only be satisfied by specifying a separate task with one or more methods that should make q true. In the block-world domain, the top-level task `achieve-goals` is responsible for this. In UMCP, q can be achieved through the use of the syntactic form $achieve(q)$. In the rest of plan-based HTN planners, there is no need for a separate task. The predicate can be part of the initial task network.

- In SHOP2, the recursive tasks (*e.g.*, `find-movable`) need an additional method which encodes the base case whose decomposition does nothing. In plan-based HTN planners, such method is not necessary, because the phantomisation takes care when a certain predicate is already achieved and nothing needs to be done.

- In SHOP2 and UMCP, there is a need for an explicit check of deleted-condition interaction. In the block-world domain of SHOP2, deleted-condition interaction is handled by using Horn clauses to reason about stacks of blocks. In UMCP, this can be accomplished generally by using $achieve(q)$, which constraints q to be true right after accomplishing the corresponding task. The rest of plan-based HTN planners do not use the domain knowledge to handle the



(a) Number of knowledge and keyword symbols.



(b) Number of domain elements.

Figure 4.2: Quantitative perception of the block-world domain in hierarchical planners.

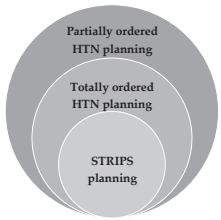
deleted-condition interaction, but instead use their problem-solving mechanisms to solve it (see Section 4.3.2).

- SHOP2 uses special so-called book-keeping operators to keep track of what needs to be done during planning. For the block-world domain, SHOP2 uses two book-keeping operators (`assert` and `remove`) that are not part of the actual block-world domain. On the other hand, other HTN planners do not use any special operators.

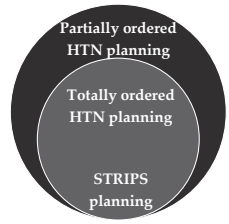
Expressiveness

We gain a perspective in theoretical expressiveness by summarising the findings in (Erol et al. 1996).[3] Figure 4.3a depicts the model-theoretic expressiveness. From this aspect, the HTN language is strictly more expressive than the STRIPS language, but totally ordered HTN planning is less expressive than partially ordered

[3]We assume that the reader is familiar with model-theoretic, operational and computational-based expressiveness. Otherwise, we refer to (Erol et al. 1996) for details.



(a) Hierarchy of model-theoretical and operational expressiveness.



(b) Hierarchy of computational-based expressiveness.

Figure 4.3: Hierarchies of expressiveness.

HTN planning, and strictly more expressive than STRIPS planning (Nau et al. 1998). An HTN planning problem (with totally or partially ordered task networks) can be transformed into a STRIPS planning problem, but the converse is not true. On the other hand, a totally ordered HTN planning problem can be transformed into a partially ordered HTN planning problem, but the converse is not true.

Figure 4.3a shows that the operational aspect has the same expressiveness hierarchy as the model-theoretic aspect. That is, HTN planning is strictly more expressive than STRIPS planning, and totally ordered HTN planning is strictly between STRIPS planning and partially ordered HTN planning. Figure 4.3b depicts the computational-based hierarchy. Once more, HTN planning is strictly more expressive than STRIPS planning. In particular, there is a (polynomial) transformation from STRIPS planning to HTN planning, but there is no computable transformation from HTN planning to STRIPS planning. Intuitively, HTN elements can represent computationally more complex problems than STRIPS operators. However, these results are true when partially ordered task networks are allowed. In fact, totally ordered HTN planning is at the same level of expressiveness as STRIPS planning, but significantly less expressive than partially ordered HTN planning. This is because totally ordered HTN planning avoids interleaving of tasks from different compound tasks.

Table 4.4 illustrates the practical expressiveness of the planning languages of HTN planners. SIPE-2, UMCP, SHOP2 and SIADEX employ the set of logical connectors ($\wedge$, $\vee$, $\Longrightarrow$, $\neg$) in task preconditions with some restrictions, while $\vee$ is not allowed in the effects of tasks. Thus, task preconditions are more expressive than the task effects. Given a predicate q, except Nonlin and SHOP2, which use deletion of q, other HTN planners use $\neg q$ in the effects. SHOP2's and SIADEX's languages

support about the level 2 of the PDDL version 2.1 (*i.e.,* numeric extensions), and allow $\Longrightarrow$ in task preconditions and $\forall$ in task preconditions and effects (with different semantics, however). Furthermore, the most extensive support for typing has SIPE-2 that goes even beyond what we defined. For example, we can state that a variable must not be of a certain type. The rest of HTN planners have either very limited or no support at all for typing. With respect to extended goals, SIPE-2, O-Plan2 and SIADEX support temporally extended and deadline goals. Temporally extended goals are expressed through the use of temporal modal operators, while deadline goals expresses conditions that must be achieved at a specific point in time in the trajectory. Default mechanisms of HTN planners do not support preferences, but SIPE-2 and SHOP2 have been extended to handle some forms of preferences. SIPE-2 is extended to two forms of preferences (Myers 1996, 2000). The first form prescribes or prohibits the use of some variables within a task, while the second one prescribes or prohibits the use of a particular task when accomplishing some objective. SHOP2 is extended to support three types of preferences (Sohrabi and Mcilraith 2008, Sohrabi et al. 2009). The first type are basic constructs of linear temporal logic. The second type are constrains, such as the precedence constraint `before(t,t')` and state constraints `holdBefore(t,q)`, where `t` and `t'` are tasks and `q` is a predicate. The most interesting type are the preferences over how tasks are decomposed into task networks (*e.g.,* prefer to apply a certain method over another), preferences over the parametrisations of decomposed tasks (*e.g.,* preferring one task grounding to other[4]), and many temporal and non-temporal preferences over task networks.

Finally, SIPE-2 and O-Plan2 support our resource hierarchy completely. In addition to the hierarchy, O-Plan2 allows sharing reusable resources unary, where a sharable resource cannot be shared among many tasks at the same time, or simultaneously, where a resource can be shared among many tasks without any specific control. SIPE-2 offers a limited mechanism for temporal reasoning, but full support can be achieved by using an external temporal reasoning system, such as Tachyon (Stillman et al. 1993). O-Plan2 appears to have the most comprehensive support for temporal reasoning compared to its predecessors (Tate, Drabble and Kirby 1994), however, this cannot be easily confirmed from the literature. SHOP2 does not explicitly reason about time. The planner has been temporally enhanced in several studies (Nau et al. 2003, Yaman and Nau 2002, Goldman 2006), but at the expense of degrading its performance and soundness. On the other hand, SIADEX provides clear explanations about its temporal reasoning mechanism and supports all relations defined independently by Allen (Allen 1983) and van Benthem (Benthem 1991).

[4]For the sake of completeness, the default mechanism of SHOP2 supports a similar feature called 'sort-by' that sorts variable bindings according to some criteria (*e.g.,* ascending order).

Table 4.4: Practical expressiveness of HTN planners ("P", "E", "C", and "ψ" stand for "preconditions", "effects", "conditions", and "constraints", respectively).

Property	NOAH	Nonlin	SIPE-2	O-Plan2	UMCP	SHOP2	SIA-DEX
Conjunction	✓	✓	✓	✓	✓	✓	✓
Disjunction	P	×	P	ψ	P	P	P
Negation	E		P, E	E	P, ψ, E	P	P
Implication	×	×	×	×	×	P	P
Existential q.			P, C		ψ		
Universal q.	×	×	E	×	×	P, E	P, E
Sort hierarchy	×	×	★★★	★	×	×	★
Extended goals	×	×	★	★	×	×	★
Preferences	×	★	★★	★	★	★★★	×
Resource	×	×	★★★	★★★	×	×	×
Time	×	×	★	★★	×	★	★★★

Competence

Table 4.5 summarises the properties related to the competence of HTN planners. We begin with the property of solution flexibility. The result of the planning process in plan-based HTN planners is a partially ordered plan which is in compliance with the definition of flexibility. An exception to this is the UMCP planner which restricts the tasks of the solution to be totally ordered. With respect to state-based HTN planners, there are two cases as well. SHOP2 produces a totally ordered plan, while SIADEX is able to plan for a flexible solution.

Except for Nonlin and O-Plan2, all planners implement DFS as their main algorithm, however, not all of them backtrack to all alternative points. NOAH, Nonlin, SIPE-2 and O-Plan2 all make use of heuristics to guide their search for a solution. During the search, Nonlin employs dependency-directed backtracking – it backtracks on choices of variable bindings and choices of task orderings, while SIPE-2 backtracks chronologically, and uses the heuristics to limit backtracking points to alternative tasks and variable binding choices only in two places. SIPE-2 does not backtrack the addition of ordering constraints. O-Plan2 uses heuristics over its choices of tasks in the plan space for which an evaluation function based on the opportunistic merit of the state is used. In addition, user interaction addresses some decision-related issues that are beyond the capabilities of the algorithms. Among

Table 4.5: Competence of HTN planners ("P" and "C" stand for "preconditions" and "conditions", respectively).

Property	NOAH	Non-lin	SIPE-2	O-Plan2	UMCP	SHOP2	SIA-DEX
Algorithm	DFS	BFS	DFS	HS	DFS	DFS	DFS
Backtracking	×	DD	Chronological				
Heuristics	✓	✓	✓	✓	×	×	×
Domain-specific control	×	C	P	C	C	P	P
Interactive control	×	×	✓	✓	✓	×	✓
Constraint management	✓	✓	✓	✓	✓	×	✓
Alternative algorithms	×	×	×	×	BrFS, BFS	IDS	×
Solution flexibility	Flexible				Total		Flexible
Completeness	×	✓*	×	✓*	✓	✓	
Soundness	×	×	×	×	✓	✓	
Execution monitoring	★	×	★★★	★★★	×	×	★★★
Replanning	★	×	★★★	★★★	×	×	★★★

HTN planners, SIPE-2, O-Plan2, UMCP and SIADEX provide user interfaces for guidance purposes. Backtracking points indeed ensure completeness, however, only UMCP and SHOP2 are provably complete and sound planners (Erol et al. 1994c). Nonlin and O-Plan2 are designed but not proved to be complete with respect to the provided domain knowledge.[5]

With respect to the ability of HTN planners to monitor the execution of plans, recognise faults, and handle them accordingly, we abstract away the mechanisms of SIPE-2, O-Plan2 and SIADEX in a general process consisting of the following components:

- **Planner** takes in the current HTN planning problem $\mathcal{P}$ and computes a solution plan sol.
- **Execution and Monitoring** takes sol and executes the actions one by one to the environment. It also processes the observations of the actual resulting state by comparing them with the expectations made during planning. If some discrepancy is recognised, a fault f is generated.

[5] Austin Tate, personal communication, November 23, 2012.

Table 4.6: Runtime and scalability of UMCP and SHOP (upper part), and SHOP2 and SIADEX (lower part).

Domain	Property	UMCP	SHOP
UM Translog	Runtime	10 problems $>$ SHOP	100 problems $>$ UMCP
	Scalability		$>$UMCP
Domain	**Property**	**SHOP2**	**SIADEX**
Zeno travel	Runtime	2 problems $=$ SIADEX	18 problems $>$ SHOP2
	Scalability	$=$ SIADEX	

- **Repair and Replanning** takes f and tries to repair the current plan. If the component is able to do so, it sends the repaired plan for execution, otherwise it may ask a user for help, and if that does not work, it asks the planner to re-plan given a modified HTN planning problem $\mathcal{P}'$.

Computation

To the best of our knowledge, three pieces of evidence report on performance and pairwise comparison results. The first evidence compares UMCP and SHOP under loads of different problems (Nau et al. 2000). The experiments are based on the UM Translog (Andrews et al. 1995), a domain similar, but quite larger than the standard logistics domain. For this domain, a set of problems with increasing number of boxes to be delivered is randomly created. The results show that the run time for UMCP is several orders of magnitude larger than the run time for SHOP. Only in first ten problems UMCP appears to perform better than SHOP, as depicted in the left-hand part of Table 4.6. Additionally, UMCP faced some difficulties when trying to find solutions to the problems. The planner tries to solve only 37% of total number of problems, and failed 45% of those 37%. The reasons for such behaviour are due to running out of memory, inability to find an answer within a specific time frame, or returning a failure.

The second evidence compares the performances of SHOP2 and SIADEX (Castillo et al. 2005, 2006). The planners are tested on the Zeno travel domain (Long and Fox 2003) under a set of different temporal problems. In all cases, SIADEX outperforms the temporal version of SHOP2 (Nau et al. 2003). The right-hand part of Table 4.6 summarises the results of this comparison.

The third evidence compares the performances of SIPE-2 and Nonlin (Wilkins 1991). The planners are compared in the domain of house construction (Tate 1976) in which SIPE-2 has four times better planning time than Nonlin for the same planning problem.

HTN planners, especially plan-based HTN ones, report obscure results about their performance. To the best of our knowledge, performance results for SIPE-2 in various domains are reported only in (Wilkins 1991). The runtime of the planner varies from one second up to six minutes for a "typical problem" in each domain. We also know that SIPE-2 is able to handle a domain that includes up to 200 tasks, 500 objects with 10 to 15 properties per object, and a problem that includes a few thousand predicates (Wilkins and Desjardins 2001).

Applicability

We define a set of domains based on the set of applications we found in the literature. The purpose of domains is to help us cluster together applications with commonalities. Some domains may not be mutually exclusive, but, for simplicity, we dispose an application only to one domain. Table 4.7 shows the list of domains where each state-of-the-art HTN planner is applied. SHOP2 and O-Plan2 are the most widely used planners, SIPE-2 and SIADEX have also a relatively high number of applications. SIPE-2 is used in at least seven applications ranging from air-campaign planning, crisis management and logistics, mission planning and oil spills, to beer production. SIADEX is employed in at least nine applications ranging from business process management, fire forest fighting, e-learning, oncology treatment, planning tourist visits and Web service composition, to planning in ubiquitous computing. O-Plan2 is used in least 14 applications ranging from air-campaign planning, biological pathway discovery, house and space platform construction, crisis management and logistics, mission planning, production and project planning, to service composition. SHOP2 is used in at least 16 applications ranging from crisis management and logistics, location-based services, plan and goal recognition, production and project planning, mobile robot planning, Web service composition, to planning in ubiquitous computing and video games. For more details on how HTN planning is applied to most of these domains, we refer to (Georgievski and Aiello 2015a).

Figure 4.4 shows the applicability of HTN planners from a time perspective. We consider as a time point of a specific application the year when its publication has appeared, or when a reference about the application is made in some publication. The majority of applications were developed in the decade 1990-2000, the time when O-Plan2 was in its prime, and the decade 2000-2010, the time when SHOP2 was in its heyday. In the most recent times, two applications are implemented by SHOP2, and three applications by SIADEX.

Table 4.7: Applications of HTN planners in diverse domains.

Domain	NOAH	Nonlin	SIPE-2	O-Plan2	UMCP	SHOP2	SIADEX
Air-campaign planning			(Lee and Wilkins 1996)	(Tate et al. 1998)			
Biological pathways				(Khan et al. 2003)			
Business process management							(González-Ferrer et al. 2013)
Construction planning		(Tate 1976)	(Wilkins 1991)	(Currie and Tate 1991, Aarup et al. 1994, Drabble et al. 1997)			
Crisis management and logistics		(Tate 1976)	(Wilkins and Desimone 1992)	(Tate et al. 1996, Kingston et al. 1996, Tate, Dalton and Levine 2000, Tate and Dalton 2003)		(Muñoz Avila et al. 1999, Gancet et al. 2005, Nau et al. 2005)	(Asunción et al. 2005)
E-learning							(Castillo et al. 2010)
Geosocial networking						(Nau et al. 2005)	
Healthcare							(Fernández-Olivares et al. 2008, Sánchez-Garzón et al. 2013)
Mission planning			(Wilkins 1988, Wilkins et al. 2003)	(Drummond et al. 1988, Tate, Levine, Jarvis and Dalton 2000)			
Equipment configuration			(Agosta 1996)				
Plan recognition						(Blaylock and Allen 2005, 2006)	
Production planning	(Sacerdoti 1975a)		(Wilkins 1991)	(Tate 1994)		(Nau et al. 2005)	
Project planning		(Tate 1976)		(Tate, Drabble and Kirby 1994)		(Muñoz Avila et al. 2002)	
Tourism planning							(Castillo et al. 2008)
Mobile robot planning		(Tate 2007)				(Weser et al. 2010, Marco et al. 2013)	
Service composition		(Tate and Lesley 1982)		(Uszok et al. 2004)		(Wu et al. 2003)	(Fernández-Olivares et al. 2007)
Software system integration						(Nau et al. 2005)	
Ubiquitous computing						(Ha et al. 2005, Marquardt et al. 2008, Song and Lee 2013)	(Hidalgo et al. 2011, Sánchez-Garzón et al. 2012)
Video games						(Kelly et al. 2008)	
Total applications	1	5	8	14	0	**16**	9

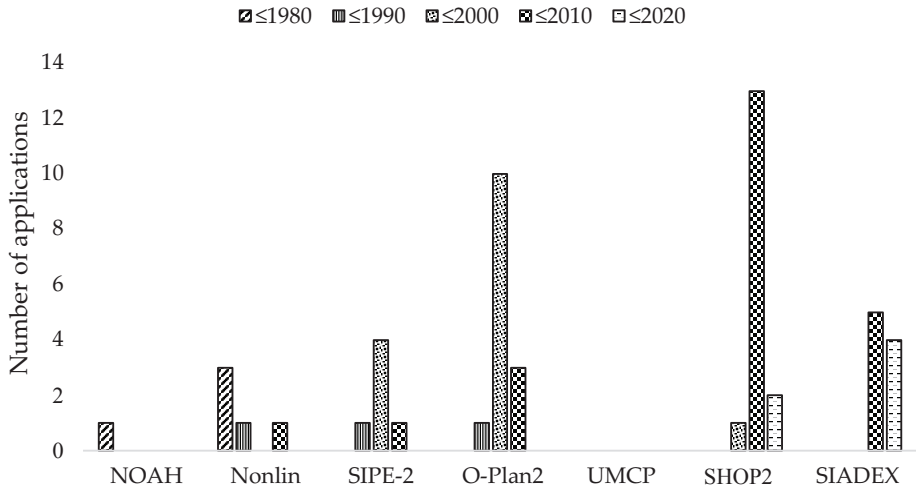


Figure 4.4: Applicability of HTN planners from a time perspective.

4.5 Remarks

For the discussion in Section 4.4.6, we could not assume that the domain models of HTN planners have the same level of expressiveness. It is therefore not possible to make statements about the relative effort needed for domain authoring.

From the theoretical discussion in Section 4.4.6, we may agree with the statement that HTN planning is able to express a broader and more complex set of planning problems than STRIPS planning. At the same time, we may question the generosity of this statement due to the assumption that the language of HTN planning uses an infinite set of symbols to represent tasks. This assumption cannot be practically supported by any planner unless restrictions are imposed (Lekavý and Návrat 2007).

Chapter 5

Reinforcing state-based HTN planning

Numerical variables are useful in many real-world domains, including ubiquitous computing (see Section 2.3.2). While many planning techniques provide support for such variables and numerical expressions (*e.g.*, (Hoffmann 2002, Kaldeli et al. 2012)), the model of state-based HTN planning defined in Section 4.2.2 is based only on logical expressions. In contrast, the state-based HTN planners, SHOP2 and SIADEX, support numerical reasoning. The main issue with these planners in this context is that their underlying planning models are unclear. The ambiguity in SHOP2 lies in the constructs and expressions that are allowed to be used in parameters, preconditions and effects of tasks. In addition, one can use structures, such as programming-level lists, mechanisms that represents goal agendas, and other solution-oriented structures to encode SHOP2 domain knowledge. Such encodings most often exhibit behaviours that go beyond the ones expected to be provided to AI planners. SIADEX, on the other hand, supports specifying domains in the manner of PDDL. While there was an attempt to define a common syntax for describing hierarchical domains in PDDL1.0 (McDermott et al. 1998), the differences between planners based on hierarchical planning undermined the efforts at standardisation and this part of the PDDL language has never been successfully explored. However, SIADEX is based on the Hierarchical Planning Definition Language (HPDL) notation (Castillo et al. 2006, Georgievski 2013), which extends PDDL with hierarchical constructs, but for which there is no well-defined semantics.

Domains provided to state-based HTN planners generally contain well-conceived knowledge. Recall that SHOP2 requires a larger and more elaborated domain model for solving block-worlds planning problems than plan-based HTN planners do (see Section 4.4.6). Among the reasons for this is the need for handling phantomisation explicitly in the encodings of compound tasks. Phantomisation is what happens after the process of recognising that some task is already accomplished by other task(s) at some place in the task network (see Section 4.3). Thus, a state-based HTN planner can find that some effect has been achieved only if the given domain knowledge contains a phantom method (or several phantom methods). Such conceived knowledge however depends on the domain author's ability and experiences to identify and encode phantomisation.

Looking at real-world domains, they are characterised by unpredictability of alternatives encountered during planning which relates to risk. The sensitivity of planning techniques to risk is especially useful for applications with large wins or losses of resources, such as money, power, equipment, time, and even humans. If a planning technique is indifferent towards risk, the result may be an undesirable outcome. To take the case of ubiquitous computing, consider a building confronted with an emergency situation due to fire. The building needs to react in such a way that all its occupants will be rescued in as short a time as possible. Let us assume that the building can accomplish the deliverance in two ways. One way involves guiding occupants through a very fast track but with a high risk of encountering flames on the route to the exit. Otherwise, the planner may choose a longer track: through this track occupants can be guided to escape the building with a lesser chance of danger of fire breaking out. Which track to decide on?

We consider all these issues by improving state-based HTN planning with concrete features. We define a domain theory that supports numerical expressions in preconditions of tasks, and operations on numeric-state variables in the effects of primitive tasks. With this, we clarify what can be included in domain encodings, define in essence the notations of HPDL, and bring closer state-based HTN planners to using a unified representational model. We then introduce an approach for automatic phantomisation in state-based HTN planning. We transfer some of the domain solution-related knowledge into the planning process itself with the goal of keeping the domain knowledge as simple and compact as possible. Finally, we use HTN planning for risk-sensitive planning domains. In fact, we propose a novel framework, called *utility-based HTN planning*, in which utility functions reflect the risk attitude of compound tasks. We also adapt a best-first search algorithm to take such utilities into account.

5.1 Numerically extended state-based HTN planning

We extend the formalism for state-based HTN planning (see Section 4.2.2) to support numeric-state variables. All sets in the following definitions are finite and non-empty, if not stated otherwise. To recapitulate, a primitive name is an expression of the form $t_p(\tau)$, where t_p is a primitive symbol, and $\tau = \tau_1, \ldots, \tau_n$ are terms. A compound name is defined similarly. We refer to the set of primitive and compound names as a set of task names T_n.

5.1 Definition (Domain theory). *A* domain theory $\mathcal{D}$ *is a tuple* $\langle \mathcal{Q}, \mathcal{V}, \mathcal{T} \rangle$*, where:*

- $\mathcal{Q}$ *is a set of predicates.*
- $\mathcal{V} = \{v_1, \ldots, v_n\}$ *is a set of variables. Each* $v_i \in V$ *ranges over a finite domain* D_i*.*

- $\mathcal{T}$ *is a set of tasks. The set of tasks* T *consists of a set of primitive and a set of compound tasks, where:*
  - *A primitive task (also an operator)* o *is a tuple* $\langle t_p(o), pre(o), eff(o)\rangle$*, where:*
    - $t_p(o)$ *is a primitive name,*
    - $pre(o)$ *are preconditions. A* condition *is a pair* $\langle p(cond), nc(cond)\rangle$*, where* $p(cond) \subseteq Q$ *is a set of predicates and* $nc(cond)$ *is a set of numerical constraints. A* numerical constraint nc *is a tuple* $\langle exp, \circ, exp'\rangle$*, where exp and* exp' *are expressions, and* $\circ \in \{<, \leq, \geq, >\}$ *is a relational operator. An* expression exp *is an arithmetical expression constructed by using a binary operator* $\diamond \in \{+, -, *, /\}$ *over* V *and* $\mathbb{Q}$*.*
    - $eff(o)$ *are effects. An* effect *is a pair* $\langle p(eff), nc(eff)\rangle$*, where* $p(eff) \subseteq Q$ *is a set of predicates, and* $nc(eff)$ *is a set of numerical effects such that for all* $\langle v_i, assign, exp\rangle$,$\langle v_j, assign, exp'\rangle \in nc(eff)$ *it holds* $i \neq j$*. A* numerical effect *is a tuple* $\langle v_i, assign, exp\rangle$*, where* $v_i \in V$*,* $assign \in \{:=, + =, - =, * =, / =\}$ *is an assignment operator, and exp is an expression.*
  - *A* compound task t *is a pair* $\langle t_c(t), M_t\rangle$*, where* $t_c(t)$ *is a compound name, and* M_t *is a set of methods. A* method m *is a pair* $\langle pre(m), tn(m)\rangle$*, where* $pre(m)$ *are preconditions and* $tn(m)$ *is a task network. A* task network tn *is a pair* $\langle TN, \prec\rangle$*, where* $TN \subseteq T_n$ *are task names, and* $\prec$ *is the order of task names in* TN*.*

A state s is a pair $\langle p(s), v(s)\rangle$, where $p(s) \subseteq Q$ is a set of predicates and $v(s) = (v_1(s), \ldots, v_n(s)) \in \mathbb{Q}^n$ is a vector of rational numbers. Basically, $p(s)$ represent predicates that evaluate to true, and each $v_i(s)$ is the value of v_i. We say that a numerical constraint is *satisfied* in state s if the value of *exp* in s is in relation $\circ$ to the value of *exp*′ in s. A numerical effect is *applied* in a state s by updating the value of v_i in s with the value of *exp* in s using the assignment operator *assign*. A primitive task o is *applicable* in a state s if $s \models pre(o)$. $s \models pre(o)$ holds iff $p(pre(o)) \subseteq p(s)$ and all numerical constraints in $nc(pre(o))$ are satisfied in s. Applying o to s results into a new state $s[o] = s \cup eff(o)$, where $p(s[o]) = p(s) \cup p(eff(o))$ and $v(s[o])$ is the vector of values generated by applying all numerical effects in $nc(eff(o))$ (unaffected values are left unchanged). A method m is *applicable* in a state s if $s \models pre(m)$. Akin to a primitive task, $s \models pre(m)$ holds iff $p(pre(m)) \subseteq p(s)$ and all numerical constraints in $nc(pre(m))$ are satisfied in s. Given a compound task t and a method m such that $m \in M_t$, applying m to s results into a task network $tn(m) = (s[m], t)$.

Having the definition of the domain theory and the semantics, we can define the HTN planning problem and its solution.

5.2 Definition (HTN planning problem). *An* HTN planning problem $\mathcal{P}$ *is a tuple* $\langle \mathcal{D}, s_0, tn_0\rangle$*, where* $\mathcal{D}$ *is the domain theory,* s_0 *is an initial state, and* tn_0 *is an initial task*

network.

A sequence of primitive tasks $o_1, \ldots, o_n$ is *applicable* in s if there are states $s_0, \ldots, s_n$ such that $s_0 = s$ and o_i is applicable in s_{i-1} and $s_{i-1}[o_i] = s_i$ for all $1 \leq i \leq n$.

5.3 Definition (Solution). *Given an HTN planning problem $\mathcal{P}$, a* solution *to $\mathcal{P}$ is a sequence of primitive tasks $o_1, \ldots, o_n$ applicable in s_0 by decomposing tn_0.*

5.2 Phantomisation

Phantomisation is a stage after the process of recognising that the purpose of some task is already accomplished by other task(s). It involves a substitution of an element of a plan with a "phantom" element to indicate that the step is a no-op or not needed in the given situation. The phantomisation is mainly used to avoid redundant, unnecessary steps in plans. In many cases, the phantomisation can be key to a satisfactory performance of HTN planners. Thus, the advantage of using phantomisation is the planner's ability to take into account which and when tasks are really necessary, and therefore, to produce more efficient plans. In most plan-based HTN planners, the phantomisation process is accomplished automatically during the planning process. This is not the case with state-based HTN planners in which phantomisation information is provided in the domain knowledge. The weak points of this type of phantomisation are its identification and encoding, and along with that, an increased domain size.

We address the possibility of diminishing the strenuous and tedious process of writing effective domain knowledge by introducing enhanced reasoning in state-based HTN planners so as to recognise and handle phantomisation automatically and without explicit domain encodings. The reasoning is performed by checking whether a current task's effects are already achieved by other same-named tasks and are still holding in the current state. In the case when these effects are still holding, a planner can reason that this task is redundant and avoid its application. By performing automatic phantomisation, a planner has to control better the search space as the phantomisation can happen at different levels of task decomposition (and interleaving) which may lead to redundant searches or plans, if such exist.

5.2.1 Approach

Our approach to automatic phanotmisation includes a notion of agenda. Intuitively, given an HTN planning problem $\mathcal{P}$, the agenda contains all atoms that are valid up to the i-th state. An atom may not be in the s_i state, but it may still be valid. That is, an atom is *phantom* in state s_i if and only if its value eval-

uates to true in some state s_{i-k} and whose truthfulness holds between s_{i-k} and s_i but not in s_i. For example, let r_1, r_2, r_3 be rooms in Theodore's home, Tars is in room r_1, and c is a cup also in r_1. Let r_1 be adjacent to r_3 and r_3 to r_2. The goal then is to move Tars from r_1 to r_2 and transfer c from r_1 to r_3, that is, *move-robot*$(Tars, r_1, r_2) \wedge$*transfer-cup*$(c, Tars, r_1, r_3)$. If we move Tars from r_1 to r_3, the fact $(at\ \ Tars\ \ r_1)$ is no longer true in the state, but it is a valid fact for the *transfer-cup*$(c, Tars, r_1, r_3)$. Recording this type of fact validity enables the task to be further decomposed by unloading the cup at r_3 and moving Tars to r_2.

5.4 Definition (Agenda). *Let P be an HTN planning problem and s_i the current state. An agenda is a set $A \subseteq s_0 \cup \cdots \cup s_i$ such that all logical atoms in A are phantoms in s_i.*

We can now check when some operator is already accomplished during the planning process. We consider such operator to be *matchable* to some already applied operator if and only if they represent the same primitive task.

5.5 Definition (Phantom primitive task). *Let $\mathcal{P}$ be an HTN planning problem and A the agenda. A primitive task t is phantom if and only if there exist another primitive task t' in the current plan π such that $t(t_p(\tau)) = t'(t_p(\tau))$ and $eff(t') \in A$.*

In addition, we need to check methods of a compound task for their applicability. Without phantomisation, the methods' precondition may not be applicable as certain bindings do not exist explicitly in the current state. In our case, relevant methods are those that have been already successfully instantiated (for example, their primitive tasks, if any, are part of the potential plan).

5.6 Definition (Phantom compound task). *Let $\mathcal{P}$ be an HTN planning problem and A the agenda. Let t be the current compound task, m its method, and s_i the current state. Task t is phantom if t is decomposable in m and m is applied to state s_k such that $k < i$ and $pre(m) \in A$.*

Algorithm 2 implements phantomisation taking as input an HTN planning problem $\mathcal{P} = \langle s_0, tn_0, \mathcal{D} \rangle$, an agenda A initialised to s_0, and an empty set $applied$ that will contain applied tasks. The main procedure starts with an interleaving step represented through the getTask procedure. This procedure prunes each primitive task that is already applied (*i.e.,* a step in the potential plan) and its effects are elements of A (according to Definition 5.5).

Planning continues depending on whether the chosen task is primitive or compound. If the chosen task t corresponds to a primitive task, which is applicable in the state s_i, its effects are added to the list of valid logical atoms A. When the chosen task t is compound, a typical state-based HTN planner (*e.g.,* JSHOP2) skips task's methods for which bindings do not exist. However, with phantomisation, we may

Algorithm 2 Planning with phantomisation

```
1:  procedure SEARCH(P, A, applied)
2:      t ←GETTASK(tn_i, A)
3:      if t is primitive then
4:          if t is applicable in the current state s_i then
5:              apply t to s_i
6:              add t to applied, update A with t's effects, remove t from tn_i
7:              SEARCH(P, A, applied)
8:      else if t is compound then
9:          if t has a method m in applied and pre(m) are in A then
10:             add m's tasks to tn_i and SEARCH(P, A, applied)        ▷ t is phantom
11:         if t is decomposable through a method m' in the current state s_i then
12:             decompose t into m' and add tasks of m' to tn_i
13:             add m to applied and SEARCH(P, A, applied)
14: end procedure

15: procedure GETTASK(tn, A)
16:     choose a task t from tn                  ▷ With respect to ordering constraints
17:     if t is primitive then
18:         if same-named operator t' is in applied and t's effects are in A then
19:             GETTASK(tn \ t, A)                                    ▷ t is phantom
20:         else
21:             return t
22:     else
23:         return t
24: end procedure
```

also consider some of those methods, in particular, those already successfully applied. Therefore, in lines 9-10, we add logic to the algorithm that handles this type of methods and call the procedure recursively with methods' tasks added to the current task network.

An HTN planner employing POTD or UTD produces all possible combinations of task sequences for a given task network (see Section 4.3.1). Previously identified phantomisation results in additional interleaving steps between tasks in the task network. But, since the task has already been accomplished, many of the interleaving steps are not necessary – they produce redundant searching. For instance, if a planner finds a plan at some point after successful phantomisation and backtracks

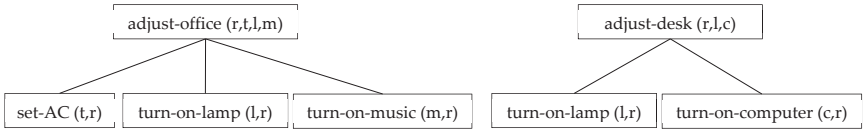


Figure 5.1: Examples of two compound tasks.

to try other combinations without controlling the search, it will find a number of plans which are equivalent to the first found one. Hence, we add a control ability to the algorithm that prunes these steps. The algorithm continues to search for other possible methods of t for which appropriate bindings exist in the i-th state. Thus, when a certain task is decomposable in s_i, its applicable method is added to *applied*.

5.2.2 Example

While some performance results of this approach implemented on top of the JSHOP2 planner are reported in (Georgievski et al. 2011), here we provide a simple example of phantomisation from a scenario in ubiquitous computing. The scenario is articulated around the adaptation of an office equipped with an air conditioning system, lamp, radio, and computer. This information can be encapsulated in two compound tasks as shown in Figure 5.1. One task is for adapting the office to certain ambience (that is, *adjust-office*), and the other one is for adjusting the work desk (*adjust-desk*). The *adjust-office* task can be further decomposed into a network of three tasks, namely *set-AC* for setting the air conditioning system to a temperature level t, *turn-on-lamp* for switching on the lamp l, and *turn-on-music* for controlling the radio r. The *adjust-desk* task contains a decomposition of two tasks, namely *turn-on-lamp* and *start-computer* for invoking the computer c. *set-AC*, *turn-on-lamp*, *turn-on-music*, and *start-computer* are all primitive tasks.

We may also have some constraints on the order of tasks in the task networks of the compound tasks, for example, that *set-AC* must occur before *turn-on-lamp* and the latter one must occur before *turn-on-music*. Thus, the tasks of *adjust-office* should be accomplished in the sequence given in Figure 5.1.

Figure 5.2 shows the encodings of the compound tasks for the planning problem just described in HPDL. `adjust-desk` and `adjust-office` tasks do not contain any preconditions to keep the representation as simple as possible.

Say that we want to perform both tasks, *i.e.*, an initial task network of preparing the work desk in the office r and adjusting r after some time being empty. We assume that one of the most effective solutions would be when the air conditions are comfortable, the lamp is turned on, the music is playing, and the computer is started and ready for work. That is, the plan π = *set-AC,turn-on-lamp,turn-on-music,turn-*

```
(:task adjust-office
 :parameters (?r ?a ?l ?m)
 (:method
  :precondition ()
  :tasks (sequence (set-AC ?a ?r)
           (turn-on-lamp ?l ?r)(turn-on-music ?m ?r))))

(:task adjust-desk
 :parameters (?r ?l ?c)
 (:method
  :precondition ()
  :tasks (sequence (turn-on-lamp ?l ?r) (start-computer ?c ?r))))
```

Figure 5.2: Simplified task encodings.

on-computer.

Let us examine the situation when our algorithm is on the right way of finding such good solution. For the purpose of demonstration, we consider a straightforward application of the first two operators of the `adjust-office` task. At this point, their effects are added to the agenda. The planning process continues by interleaving the task (`adjust-desk r l c`) and decomposing it to its task network. Task's first subtask is (`turn-on-lamp l r`) which is already a part of the potential plan. The algorithm reasons that this task is already achieved and that its effect (the lamp is on) is still valid. Therefore, the algorithm is allowed to prune the task from interleaving and continues by processing the rest of available tasks. In few steps, the algorithm finds the correct sequence of operators.

In contrast to our approach, a typical state-based HTN planner will not find a solution given the domain encoding in Figure 5.2. In order for such planner to come up with a plan, we have to improve the domain with more effective descriptions. In Figure 5.3 we enclose such enhanced encodings. As we can see, we have to include an additional task `light-helper` which has a method representing phantomisation – doing nothing when the lamp is already turned on. Comparing Figures 5.2 and 5.3, one notices the first is a simpler and more compact domain representation.

5.3 Utilities

In making decisions, an individual or a system will try to enhance his utility or a given one. Utility is defined such that a preferred alternative always represents a higher utility level than the less preferred (or rejected) alternative.

```
(:task adjust-office
 :parameters (?r ?a ?l ?m)
 (:method
  :precondition ()
  :tasks (sequence (set-AC ?a ?r)
           (light-helper ?l ?r)(turn-on-music ?m ?r))))

(:task adjust-desk
 :parameters (?r ?l ?c)
 (:method
  :precondition ()
  :tasks (sequence (light-helper ?l ?r) (start-computer ?c ?r))))

(:task light-helper
 :parameters (?l ?r)
 (:method
  :precondition (not (on ?l ?r))
  :tasks (sequence (turn-on-lamp ?l ?r)))
 (:method
  :precondition (on ?l ?r)
  :tasks ()))
```

Figure 5.3: Enhanced task encodings.

5.3.1 Utility theory

Utility theory deals with decision making according to a given risk attitude under several assumptions and unlimited (planning) resources available (Neumann and Morgenstern 1947). In order to illustrate the theory on an example, we take the famous paradox of the St. Petersburg game studied by Daniel Bernoulli:

> Theodore tosses a fair coin until heads appear for the first time. If heads shows up on the k-th toss, he gets 2^k Euro. How much is Theodore willing to pay in order to be allowed to participate in this game?

Let Theodore own a wealth of w Euro. If he chooses to play the game and pays c Euro for participation, he owns $2^k + w - c$ afterwards with probability $\frac{1}{2^k}$. The expected reward of the game then is $\sum_{k=1}^{\infty} \frac{2^k+w-c}{2^k} = \infty$ Euro. Thus, irrespectively of how much the participation costs, the expected reward of playing the game is larger than the expected reward of abstaining.

Bernoulli was of the opinion that people do not average over rewards, but over the satisfaction or utility that the rewards provide. For every type of risk attitude, there exist a utility function that transforms c into utilities $u(c) \in \mathbb{R}$ such that it is

reasonable to maximise expected utility. This is done only if the decision maker (a person or a system) follows several simple preference assumptions (axioms) and has unlimited planning resources available.

$u(w)$ is the utility when Theodore is not participating in the game, and u is the utility function of the decision maker. Theodore's participation in the game implies the expected utility $\sum_{k=1}^{\infty} \frac{u(2^k+w-c)}{2^k}$. Thus, he should bet at most c_{max} Euro, where $u(w) = \sum_{k=1}^{\infty} \frac{u(2^k+w-c_{max})}{2^k}$. Theodore (and people in general) is not willing to pay more for the participation because he is risk-averse. That is, losing a part of his wealth means more to him than winning a fortune. People are risk-averse (risk-seeking, risk-neutral) if and only if their expected utility of every non-deterministic game is smaller than (larger than, equal to) their utility of the expected reward of the same game. People that are not risk-neutral are risk-sensitive (Koenig and Simmons 1994).

5.3.2 Framework

We propose a framework based on HTN planning that takes task utilities into account, where a utility is a function of a profit and may determine the attitude towards risk (Georgievski and Lazovik 2014). We assume that a primitive task is associated with a function of consumption that expresses a single or combination of properties, such as failure rate and energy use. The possible runtime failures are not modelled directly, but we assume that some tasks are more likely to show unpredictable behaviour. Under the assumption that a compound task may have a large number of methods and many of them be applicable, we use a *utility* to express the level of preference of the compound task, and a *utility function* to calculate the perceived utility of the task.

We require a primitive task to be provided with some function of consumption that expresses the cost of the operator as a non-positive value, that is, $c(o) \in \mathbb{R}_{\leq 0}$. We estimate the utility of a compound task t based on its risk or consumption attitude. We assume that each t is *acyclic*, that is, t can be decomposed only to a finite depth. The following function represents a template formula for calculating the utility value of t

$$U(t) = \begin{cases} c(o), & \text{if } t \text{ is primitive;} \\ \min\limits_{m \in M_t} E(m), & \text{otherwise,} \end{cases}$$

where E is an estimation factor. In fact, the definition of the estimation factor E gives a concrete utility function. Based on the discussion about utility theory and the types of attitude towards risk and consumption, we define four estimation factors,

Table 5.1: Utilities for the task t in Figure 5.4.

Utility function	Utility value
Risk-averse	-3
Risk-seeking	-1
Risk-neutral	-2
Consumption-aware	-5

that is, utility functions.

Risk-averse utility. A task is *risk-averse* if and only if it maximises the minimum expected utility value of its methods' subtasks. A risk-averse task shows pessimistic behaviour towards risk, that is, it represents the safest possible decision by using

$$E(m) = \min_{t' \in tn(m)} U(t').$$

Risk-seeking utility. A task is *risk-seeking* if and only if it maximises the maximum expected utility value of its methods' subtasks. A risk-seeking task shows optimistic behaviour towards risk, that is, it takes the highest risk to gain the best outcome by using

$$E(m) = \max_{t' \in tn(m)} U(t').$$

Risk-neutral utility. A task is *risk-neutral* if and only if it maximises the average expected utility value of all subtasks for a given task's method, that is,

$$E(m) = \frac{\sum_{t' \in tn(m)} U(t')}{|tn(m)|}.$$

Consumption-aware utility. A task is *consumption-aware* if and only if it maximises the sums of utility values of its methods, that is,

$$E(m) = \sum_{t' \in tn(m)} U(t').$$

In Figure 5.4 we show an example of a compound task whose leaf tasks are operators associated with non-positive costs. Applying risk attitudes to this task by using the utility functions we just defined results in the utilities values shown in Table 5.1.

We can now define the notion of *utility-based HTN planning*. To do so, we assume a utility function u for a plan π to produce an expected total utility as sum of the costs of plan's steps.

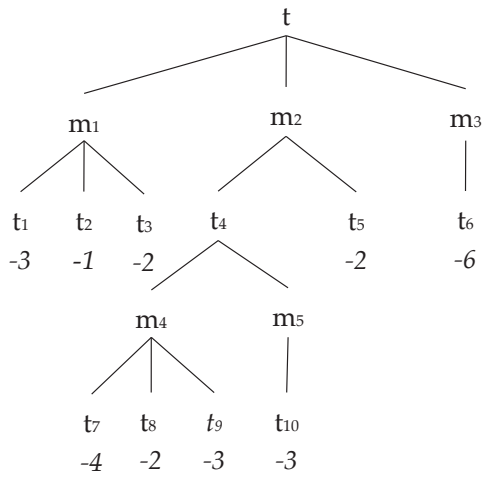


Figure 5.4: Example of a task.

5.7 Definition (Utility-based HTN planning). *An HTN planning problem $\mathcal{P}$ with utilities is a tuple $\langle s_0, tn_0, \mathcal{D}, U\rangle$, where U is a utility function. A plan π is a solution to $\mathcal{P}$ if and only if there does not exist a plan π' for $\mathcal{P}$ such that $u(\pi) < u(\pi')$.*

5.3.3 Algorithm

We propose an algorithm that selects the best task network to work with, that is, the one with the highest utility value. The utility-based best-first search algorithm shown in Algorithm 3 takes an HTN planning problem, some resource value and a utility function, and starts by setting the frontier to an initial node. We take a node n to be a three-element structure $\langle s, tn, \pi\rangle$, where s is a state, tn is a task network, and π is a partial plan. A node n is selected for expansion based on one of the utility functions that can be seen as a heuristic function. The nodes whose absolute utility value is greater than the amount of resource are pruned from the search space. If the node has no more tasks to be decomposed, and the utility of its partial plan is higher than the utility of the best plan found so far, then we consider the current one as the best plan. The function in line 10 can be one of the task decomposition styles discussed in Section 4.3.1. The chosen style must consider the applicability of operators and methods in the state of the current node.

5.8 Theorem (Optimality). *Given a utility-based HTN planning problem $\mathcal{P}$ and a resource value, if the termination of the algorithm returns a plan, then the plan is an optimal solution to $\mathcal{P}$ with respect to maximising the utility of using the given resource.*

Algorithm 3 Utility-based search for solution

Input: $\mathcal{P}$, U, $resource$
Output: best plan

```
1: currentUtility ← ∞
2: frontier ← ⟨s_0, tn_0, ∅⟩
3: while frontier ≠ ∅ do
4:     best ← POP(frontier)
5:     if resource > |U(best.tn)| then
6:         if best.tn = ∅ and u(best.π) > currentUtility then
7:             output best.π
8:             currentUtility ← u(best.π)
9:         decompositions ← DECOMPOSE(best)
10:        frontier ← MERGE(decompositions, frontier, U))
11: end while
```

The proof of the theorem follows from the definition and properties of the best-first search algorithm (Russell and Norvig 2003).

5.4 Summary

Among planning techniques we chose to work with HTN planning, and in particular, with state-based HTN planning. With the intention to make this technique even more suitable for real-world environments, such as ubiquitous computing ones, we augmented the technique with several features. We first defined the semantics of state-based HTN planning with numerical expressions. This formal model supports the syntax of HPDL. We also showed how can phantomisation be performed in state-based HTN planners automatically, and the benefits of it use. The main advantage is the simplification and reduction in size of the domain knowledge. Finally, we proposed the framework of utility-based HTN planning to associate an attitude of tasks towards the risk of using a given resource. We provided an algorithm that outputs optimal solutions, that is, plans with maximised utilities of using the resource.

Chapter 6

Planning as a service

The applicability of large systems, such as ubiquitous computing ones, is challenged by scalability, distribution, interoperability, evolution, and reusability. As part of such systems, planners inherit most of these challenges mainly due to the characteristics of problems and situations encountered in real-world domains. These include the size and diversity of planning problems, deployment location of planners, distribution of domain knowledge, mixed decision-making, *etc.* Considering ubiquitous computing, the problem size may vary depending on, for example, the proliferation of devices, planner's implementation may reside, let's say, in the home or in the Cloud, domain knowledge may be distributed centrally or to multiple components (or devices), and decision-making control based on AI and user decisions, which may lead to conflicts. On top of these concerns, the scenario of planning problems may vary between homes, office building, to hospitals.

The ability of planners to integrate in large systems is something that research in planning appears to be neutral to principally due to its orthogonality to the reasoning capabilities of planners as primary concerns of the community. A planning system is integrable if it offers its essential functionalities to interested components in a defined and standardised way, and is able to interoperate with them. Most of the current planners fail to be integrable. To the best of our knowledge, the only attempt, which considers this issue and is made only recently, focuses on the interoperability of planning systems employed in space-mission domains (Fratini et al. 2013). Also, several questions related to runtime behaviour, interoperability and scalability of planners in ubiquitous computing are raised in (Marquardt and Uhrmacher 2009c). The study however does not discuss in details the possible design of planning systems that could answer the questions.

Since the current situation witnesses planning systems with no standardisation, the obvious consequence is the strong need for familiarity with details of planning systems. The interoperability of planners is only possible under the assumption that they offer a standard interface or a set of services to other components of the underlying system. A common way to accomplish this in a transparent way is by considering the primary elements of Service-Oriented Computing (SOC) (Papazoglou and Georgakopoulos 2003, Erl 2007). The first element is the model of Service-Oriented

Architecture (SOA) which provides foundations for designing large and cooperative systems. The capabilities of the components of SOA-based systems represent application services (see Section 1.4). In order planners to be integrable in SOA-based systems, they need to take into account another primary element, namely service-orientation. Service-orientation focuses on the structure and implementation of planning functionalities. These are offered as services to other components of the system. We refer to the services that planners provide as *planning services*. Since planning services are used at the application level of the system, they represent a subset of the application services.

We propose the concept of *planning as a service*, that is, planners to be completely service oriented (to comply with SOC principles), and consequently to support easy construction of cooperating distributed systems (based on SOA). While in the following we analyse the integration of planning systems in a larger distributed architecture in the context of ubiquitous computing, our proposal is general and can be considered in a wider range of contexts. The benefits of defining planning functionalities as services can be multiple: (1) efficiency - rapid prototyping design; (2) flexibility - arbitrary system configurations, in terms of (new) planning functionalities, can be easily integrated; (3) scalability and fault tolerance; (4) interoperability; (5) reuse - usable in various ubiquitous computing domains with minor changes (tailoring), and also reduced development time, cost, training, *etc.*

Finally, we introduce the **SH** planning system, some of its engineering and implementation details considering the principles of service-orientation, and the set of planning services it supports. **SH** is used in two applications illustrated in the following chapters.

6.1 Service-orientation

A system is service oriented if the concept of service-orientation is applied to it (Erl 2007). Service-orientation affects the way of organisation and implementation of the functionalities of systems. In this regard, several design principles appear to be important but challenging for planners to achieve service-orientation.

Interfacing

We can ensure the interoperability of a planning system with other components of an architecture by considering two important design principles. The *service contract* defines interaction requirements and constraints as well as the semantic information of the planning system made available to application services consumers of planning services. In other words, this principle enforces the planner to adhere to a communication agreement, and to provide an interface with a description of what

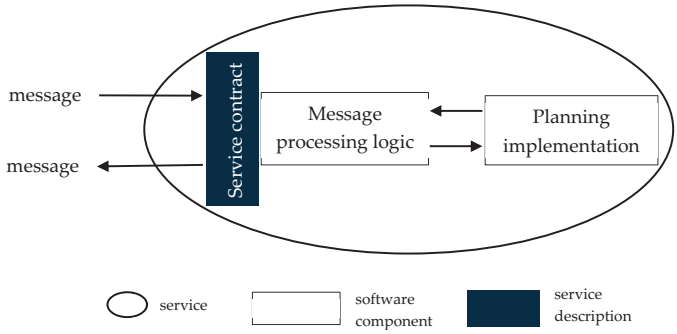


Figure 6.1: Profile of a planning service specified in the notation of (Erl 2007).

it offers and requires. The description can be specified using interface description languages, such as the RESTful Service Description Language (RSDL) (Robie et al. 2013), Web Service Description Language (WSDL) (ConsortiumW3C 2005), and so forth. The second principle is closely related to the service contract. *Service abstraction* ensures that the knowledge about the planning implementation (or domain model) offered to other components through the contract is limited (see Figure 6.1). Service contract and service abstraction should be defined in such a way that the planning system provides general but detailed enough planning services to be practically usable. This is important especially because many planning systems use domain models specific to the planning technique adopted. For example, the model of a PDDL-based or a CSP-based planner is reasonably different than the model of an HTN planner. Therefore, the level of abstraction directly affects the easiness (or equally, the difficulty) to specify a planning request or to interpret a plan. Obviously, a standard contract cannot be defined easily, at least not at the current stage of development of AI planning (many planning techniques use their own domain models, or in a better case, an extension of a more generic planning model, such as PDDL).

The main challenge with respect to interoperability is to find the state in which these two principles are in equilibrium. Let us analyse why we need such a state.

If we standardise the contract only, it means that the semantics for the planning problem must be defined within the planning system. Ergo, planning services would require a minimal specification for invocation, and would refer to predefined domain models. This case is only possible under the assumption that the relevant data needed for the planning process would be available and certain at the time of each request issued to the planning system. Practically, this assumption might be too strong for ubiquitous computing environments. These environments are highly

dynamic (in terms of context changes) and involve different types of requests, such as user preferences (see subsection Requests in Section 2.3.2). If the planning system maintains the state of the environment, then we need to assure that the system will contain the latest update of the context. Otherwise, the context information should be provided along the preferences and the predefined goal.

If we standardise the semantics, it means that the specification of relevant data needed for the planning process would be provided through the contract of the planning system. The more information is exposed in the contract, the deeper the coupling of consuming services to the contract can become (see Section 6.1). Therefore, one should have great knowledge of the planning technique adopted by the system, or there should be an agreement on some abstract domain model. The first case is what we aim to avoid in the first place, while the second one represents an ambition already existing within the AI planning community (Fratini et al. 2013).

One way to address the challenge of finding a state of balance is to agree on the contract and the level of abstraction for the most essential services that a planning system should provide, for instance, in ubiquitous computing environments.

Loose coupling

Loose coupling minimises the level of dependency between planning services and application services consumers of planning services, but also the level of dependency among planning services themselves. This means that the relationship between a planning service, its underlying environment, and its consuming services is clearly defined. In essence, a planning system would support loose coupling if the dependency of a dependent service is to the contract of the planning service and not to internal and concrete classes of the planning technique, and also if the dependency of the planning service to some outside logic is reduced as much as possible (see Figure 6.2). What could be problematic with the application of this principle is the case when contracts for planning services would be derived directly and only form the implementation of existing planning techniques, and therefore, such contracts could become tightly coupled on that existing implementation.

A planning system receives messages from other application services, such as a context update or planning request, but also publishes messages to other services, such as the component responsible for the execution of device operations (*cf.* Orchestration component in Figure 1.1). In this regards, loose coupling of the planning system can be increased if the system supports and handles some flexible file format, such as Extensible Markup Language (XML) or JavaScript Object Notation (JSON). The planning system can then provide clearly defined interfaces on how it uses transmitted data. For example, the planning system could provide a set of

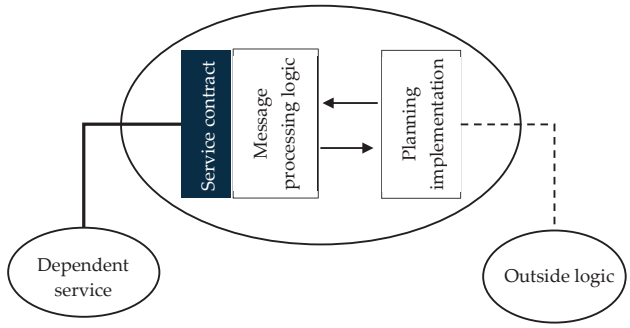


Figure 6.2: Service loose coupling specified in the notation of (Erl 2007).

methods used to extract information from received messages by exposing an XML/JSON schema.

Reusability

A planning system that adopts a domain-independent planning technique meets the opportunity to be reused in various domains. However, the planning system would be truly independent if it is not attached to any specific underlying environment (or workflow which represents the business/execution steps of the underlying system). In this context, the principle of *reusability* provides the possibility for a planning system to be reused across various technological environments and in multiple workflows, and therefore, it supports planning systems to be applied to diverse domains. Resuability of planning systems can be ensured if:

1. Each planning service is defined within an agnostic functional context. This means that the functionality that a planning service provides is associated with a context that is neutral – it does not take any specific workflow (or scenario) into account so as to be considered reusable. For example, a service providing domain modelling capability also stores domain models to some database. Its functional context is agnostic because it deals with the definition or modification of classical domain concepts only, such as state variables, constrains or actions, but the context is not about details related to the database used to store data.

2. The implementation encapsulated by a planning service is generic enough to support various situations in a single domain or different domains, where these situations are created by application services consumers of that service. In core planning services, such as domain modelling or solving a planning

problem, this characteristic is naturally supported by the domain independence typical of the majority of planners.

3. A planning service has an easily extensible contract. This characteristic gives services flexibility to deal with diverse input and output messages, that is, the format and type of messages (of course, this overlaps with the goals of the service contract principle).

4. A planning service can be accessed concurrently by other parties. Some planning services, such as solving a planning problem, might be invoked simultaneously by multiple office users, for instance. If the planning system maintains the state of the office environment, which is used by the invoked service, then the system needs to take care of concurrent use and updates of that state. Practically, concurrent computation in planning services can be realised by implementing, for example, the actor model (Hewitt et al. 1973).

Autonomy

The independence of planning services from the environment in which they are deployed, invoked and executed is ensured by the principle of *service autonomy*. The immediate benefit of autonomy is increased runtime reliability of the planning system to unexpected outside behaviour (the behaviour of other application services). The reliability of a planning service depends on the service's operational control and external resources. That is, the service needs to become more independent from the external resources. An example includes a database that stores and provides static information about a ubiquitous computing environment (*e.g.*, spatial information about an office building), which may not be available when required by some planning services.

The application of this principle includes two types of autonomy. Design-time autonomy gives the freedom to evolve planning services without affecting the way in which other application services consume them. The implementation of design-time autonomy can be guaranteed by the application of service abstraction and loose coupling principles. Runtime autonomy defines the level of control that a planning service has over its implementation when processed in a runtime environment. The application of runtime autonomy involves support for deployment of planning services to distributed environments and handling of resources required at runtime. Some planning services would face difficulties in achieving this principle as they might depend on data coming from other application services. For example, the service that solves a planning problem cannot be correctly executed if the application service that is responsible for the environment context is down and has not

provided the most recent changes of the environment. As a result, the autonomy of the planning service depends on the autonomy of the services that participate in the current workflow.

Statelessness

The scalability of planning systems can be supported to a large extent by the principle of *statelessness*. The way of managing state data, which is the information associated with the process currently being executed, determines the level of statelessness that a planning service is characterised by. In other words, this principle suggests to separate state data from a planning service whenever possible. Conversely, if the planning service actively maintains and processes state data, it constantly consumes a reasonable amount of memory and CPU usage (an even worse case is when the planning service is simultaneously accessed by multiple consuming services).

The state data that a planning service may process can represent environmental and session data. Environmental data can be static and dynamic. Static data represents the information about, lets say, the spatial organisation of an office building, and obviously, it does not actually represents the state of a planning service or the state of the current process, but it might be relevant to some capability of the service, such as solving a planning problem. Dynamic data represent the information about the events, that is, context changes that currently happen in the environment. Dynamic data is usually passed between application services. In many ubiquitous computing architectures, such as the one in Figure 1.1, the management of environmental data is deferred to a common architectural component (the Repository component in Figure 1.1). This data deferral gives the possibility to application services, including planning ones, to retrieve information when needed. As a result, the planning system may become responsible for managing the retrieved data. In practice, planning systems would retain static and dynamic data, which is retrieved from some storage on a system's first invocation, and update the dynamic data whenever a context change occurs. Practically, this means that the planning system has the latest environmental state, and it does not overload the communication by frequently retrieving a large amount of data.

The session data represents information about a connection established between a planning service and its consuming services. Statelessness says that we need to provide a possibility for a planning service to transit from an active and stateless mode to an active and state-processing mode in an efficient way. Planning services for which full statelessness is not possible (or not desirable), such as the service to solve a planning problem, should implement some form of moderate statelessness.

This means that if the planning system is idle for a long period, then its services should transit to a stateless mode.

An example of an HTN planner that supports a certain form of statelessness is O-Plan offered as a Web service (Tate, Dalton and Levine 2000, Tate and Dalton 2003).

Discoverability and composability

To search for and find services that provide the functionality we need, services need to be outfitted with meta data that will be used when the discovery search is performed. If the service can be discovered, then it possesses a degree of *discoverability*. From a planning perspective, this principle does not bring any additional challenges than those for the other application services. The application of this principle includes providing planning services' contracts with adequate meta data that will be referenced in discovery searches, and also used to communicate the functionality of those services. In addition, planning services should be able to announce their existence by registering their meta data to some service registry (*e.g.,* a distribution configuration service (Degeler et al. 2013)).

Distributed systems address larger problems by aggregating the capabilities of different services members of their underlying service-oriented architectures. *Service composability* is in favour of this concept and provides design considerations that guarantee that services can be part of diverse compositions. This enables service-oriented systems to automate their workflows by combining multiple services.

The application of service composability requires a service that is a part of a composition either as a service that controls other services, or as a service that provides functionality to other services in the composition (and does not further compose other services). The purpose of this principle is close to the goal of service reusability principle. On the other hand, the effectiveness of the principle depends on the service contract, loose coupling, autonomy and statelessness. Thus, the application of this principle can be a consequence of these principles. Their implementation should ensure, to a large extent, that planning services will exhibit a high level of composibility, and can participate in the automation of the workflow of service-oriented systems.

6.2 Services

By considering the general classification of services as suggested in (Erl 2007), the categorisation of planning services in space-mission domains (Fratini et al. 2013), and our experience, we organise planning services in the following classes.

1. *Modelling services* are entity centric because they base their functionality on planning entities that are used to form a planning problem. These services can be considered as highly reusable because they are neutral to any process. An example is a service for modelling planning domains.

2. *Problem-solving services* are task centric because they have a functional boundary associated with the task of solving planning problems. These services provide their functionality by using the capabilities of modelling services. The most important of which is the Solve planning problem service that represents the core planning process.

3. *Management services* are also task-centric ones whose functionality is related to the use and management of planning problems. For example, a service for parsing or validation of planning problems.

4. *Utility services* have a functional context that is not directly related to the core capabilities of planning systems. Utility services have a set of capabilities that handle various technological environments in a sense that make modelling and problem-solving services available and suitable for a specific system. For example, a service for storing domain models.

The first three classes of planning services depend on the adopted planning technique, that is, the implementation and language used to define domain models and planning problems. We believe that many of these services can be mapped to respective internal classes of existing planning techniques with little effort. This way, instead of standardising specific syntax or interfaces of internal classes, we can focus on the standardisation of a general set of services.

The last class of services makes the services of the first three classes suitable for a specific type or several types of runtime environments.

6.2.1 Modelling services

Planning problems are composed of predefined domains and some specific problems. The result of solving planning problems is a set of solutions or plans. For each planning entity, we envision a modelling service, that is, Domains, Problems and Plans services. The Domains service can be used in several scenarios, such as in solving planning problems, as discussed later, or in user-supportive tools for authoring domain models (see Section 6.3.2 or, for example, (Simpson et al. 2001)). The capabilities of the Domains service enable definition and manipulation of diverse concepts: state variables and/or predicates, actions and tasks, axioms and preferences. To take the case of ubiquitous computing, state variables may be used to

Table 6.1: Interface of the Domains service.

Operation	Input	Output
getDomainObject	unique domain ID	domain object
modifyDomain	unique domain ID and planning entity	domain object
addDomain	domain file	acknowledgement

represent attributes and input parameters of device services (Kaldeli et al. 2012), while predicates may be used to describe characteristics and states of objects present in the environment (Pajares Ferrando and Onaindia 2013, Kotsovinos and Vukovic 2005, Marquardt and Uhrmacher 2009a). Actions are used to update the values of state variables (Kaldeli et al. 2012), or to change the states of predicates (Pajares Ferrando and Onaindia 2013). Tasks, on the other hand, are used to encode such complex knowledge about ubiquitous computing environments that actions are not able (or not supposed) to capture. Examples include activities in the domain of elderly care (Yordanova 2011), assistance for diabetic people (Amigoni et al. 2005), adaptations of offices (Georgievski et al. 2013), *etc.* Axioms may be used to decouple device services from the rest of the knowledge about the environment (Heider and Kirste 2002). Finally, preferences are an important construct that can be used to customise the behaviour of ubiquitous computing environments according to the needs of their occupants (or users). Although preferences are not fully exploited in the setting of ubiquitous computing environments (see subsection Requests in Section 2.3.1), simple soft constraints are supported in (Kotsovinos and Vukovic 2005), for example.

Table 6.1 establishes a set of operations that could be used to define the interface of the Domains service. The set is indicative and should be further extended with more operations.

The Problems service is used to assemble the current state of the environment and the goal to be accomplished. The Plans service is used to represent the solution for a specific planning problem. A plan is represented as a collection of actions along with a set of temporal constraints to determine the order among them.

6.2.2 Problem-solving services

Solve planning problem service encapsulates the requirements of the main planning process, and fulfils them by composing the capabilities of the Domains and Problems services to acquire the domain and problem objects, respectively. Its interface can include, for example, solving for the first found plan and solving for the best plan.

Another important functionality especially for dynamic environments, such as ubiquitous computing ones, is continual planning (Kaldeli et al. 2011). The service that supports the process of interweaving planning with action execution is called Plan continually service. The service may use the capability of the Solve planning problem service whenever there is a new request issued. Otherwise, this service will try to refine the current solution by taking into account the changes of the context (that is, the initial state). If possible and supported by the planning system, dynamic changes can be incorporated into the domain object too.

6.2.3 Management and utility services

Management services encompass a set of functionalities related to parsing, validation and verification (Long et al. 2009), and visualisation of planning domains, problems and solutions (Gerevini and Saetti 2008). These services appear to be handy in ubiquitous computing environments in cases where users are empowered to interact with the environment (Butz 2010). In particular, users should be able to modify the existing domain model, for instance, by stating their preferences through a graphical user interface. Consequently, their input must be validated and verified with respect to the adopted formalism, and with respect to the referenced domain model. Hence, these services should support automatic analysis and translation of planning domains, problems and solutions. This approach will significantly reduce the need for a planning expert (or domain expert) to validate changes made by different users. Furthermore, in many types of ubiquitous computing environments, a plan synthesised by the planning system needs to be presented to a user for inspection, modification, approval, or just as advice to be followed. Visualisation services are envisioned to provide such capabilities and indeed to support the user interaction.

Utility services include functional contexts related to message conversion, storage of domain models, communication with other application services, exception handling, *etc.* Assuming that other classes of planning services are available, utility services will generally not require involvement of planning experts when being modelled. These services may include capabilities suitable for several types of ubiquitous computing environments whose implementation is based on reuse of planning services from the other three classes.

6.3 Engineering SH

We introduce the *Scalable Hierarchical (***SH***)* planning system developed to support some of the features presented here and in Chapter 5, and used in the applications discussed in the following chapters. The internal architecture of **SH** is shown

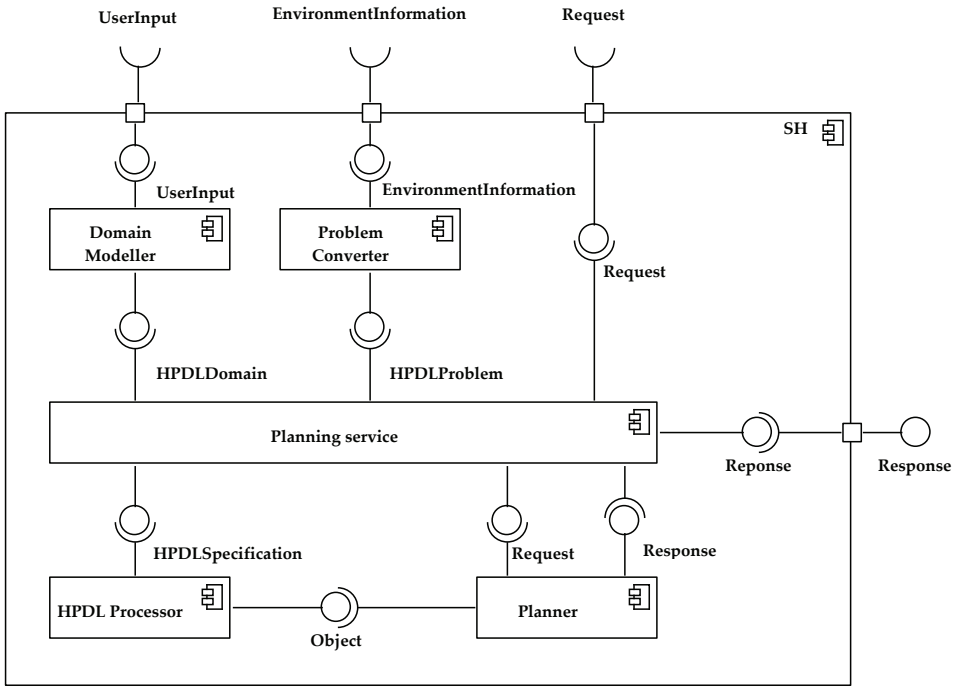


Figure 6.3: Component diagram of the **SH** planning system.

in Figure 6.3 and consists of three main components, namely Problem Converter, Domain Modeller, and Planner. The *Problem Converter* component generally accepts state information and desired objectives about some environment, and translates them into a standardised form acceptable by the Planner. This components supports several underlying message exchange and interconnection mechanisms, which enables easier integration of **SH** into large system architectures. The final result of the conversion is a complete problem description that is part of the Planner's input.

The *Domain Modeller* component is a Web-based editor that enables intuitive guidance for users when creating, viewing and modifying the domain knowledge required by the Planner component. The modeller offers an abstraction in the way of forming domain tasks and verification of the correctness of the knowledge being produced or altered with respect to the Backus-Naur Form (BNF) of the input language of the Planner. The domain models are stored and available to be used as another part of the input to the Planner.

The *Planner* component (or just the planner) is founded on state-based HTN planning. It is able to administer typing the parameters that appear in operators and tasks, conjunction, disjunction, implication and universal quantification in pre-

conditions, numeric fluents in both preconditions and effects, universally quantified effects, and axioms. Given a translation of domain and problem descriptions into an object representation, the planner uses depth-first search on the initial task network to synthesise a solution plan.

6.3.1 Syntax processing

SH performs double transformation of HTN planning problems. First, the information coming from an environment (and similarly, the knowledge coming from a user) is transformed into a problem described in HPDL through the Problem Converter (a domain described in HPDL through either the Domain modeller or a text editor). In (Georgievski 2013), we provide the syntax for HPDL using an extended BNF. The HPDL problem and domain descriptions are then transformed into programming-level constructs through the HPDL Processor (see Figure 6.3).

We design the Problem Converter upon the Cake pattern (Odersky and Zenger 2005). The basic idea behind this pattern is to solve dependencies between components through abstraction of implementation details. In particular, we define an interface for the Problem Converter through a trait that may be instantiated in multiple ways depending on the number and types of environments we need to implement. In this way, on the one hand, all components invoking the Problem Converter see and use the same set of operations. On the other hand, all further changes we make on the implementation of the converter will not affect the depending components.

The Problem Converter is implemented in the Scala programming language.[1] Scala is a type-safe language designed to be concise, logical and powerful with many implicit techniques to help simplify some common tasks. It may also improve the performance of the underlying application in terms of efficiency and scalability (when combined with other tools, *e.g.,* Akka (Wyatt 2013)).

The HPDL Processor is also implemented in Scala. We use parser combinators (Odersky et al. 2011) to transform HPDL descriptions into Scala objects. A parser is a function that takes input tokens and transforms them into programming-level constructs. A combinator is a higher order function that combines two functions into a new function. A parser combinator is then a function that combines two parsers into another parser.

Listing 6.1 shows a code snippet of parser combinators for an atomic formula and predicate: | is the alternation combinator, ~ is the sequential combinator, ^^ is a transformation combinator, and ~> (<~) is a combinator that does not include the content on its left (right) in the result. The result of the transformation for the

[1] `http://www.scala-lang.org/`

`predicate` value is the Scala case class `Predicate`.

```
lazy val atomic_formula = predicate | equality_predicate
lazy val predicate: Parser[Predicate] = "(" ~> predicate_name ~ terms <~ ")"
                                        ^^ {case pn ~ t => Predicate(pn, t)}
```

Listing 6.1: Example of parser combinators in the HPDL processor.

6.3.2 User-friendly domain manipulation

A graphical domain modeller has been designed and implemented to guide users in manipulating HPDL domain models. The modeller is also implemented in Scala, and details about the implementation can be found in (Hoekstra 2013). The Domain Modeller is a user-friendly, Web-based interface which empowers users to create new domain models, offers a graphical and textual overview of existing domains, and supports modifications and removal of domains from a domain database.

The Domain Modeller presents the users with the constructs of HPDL and shields the user from all unnecessary details of the language, such as the construct structures and naming conventions. For example, a variable must begin with a question mark directly followed by the variable's name. The name has also some expression constraints. What the user sees in the Domain Modeller is only the text box shown in the upper part of Figure 6.4.

The Domain Modeller checks and verifies the structure and expression of domain constructs regularly. If the modeller identifies an error, a message is shown to the user indicating the position of the error and possibly a user-friendly suggestion. The lower part of Figure 6.4 shows an example of the user entering an incorrect name for a domain.

The Domain Modeller assists the user in choosing the allowed constructs, and combinations of constructs and expressions. Figure 6.5 shows an example of user assistance. While the upper part contains buttons for adding precondition expressions, they are no longer visible in the lower part where the user has chosen to add an atomic formula. The modeller allows only constructs relevant to the atomic formula to be inserted by the user. Thus, shielding other precondition expressions keeps the user from building erroneous encodings.

Future modifications of existing domains do not have to be parsed because the modeller accepts a domain only if its constructs are error free.

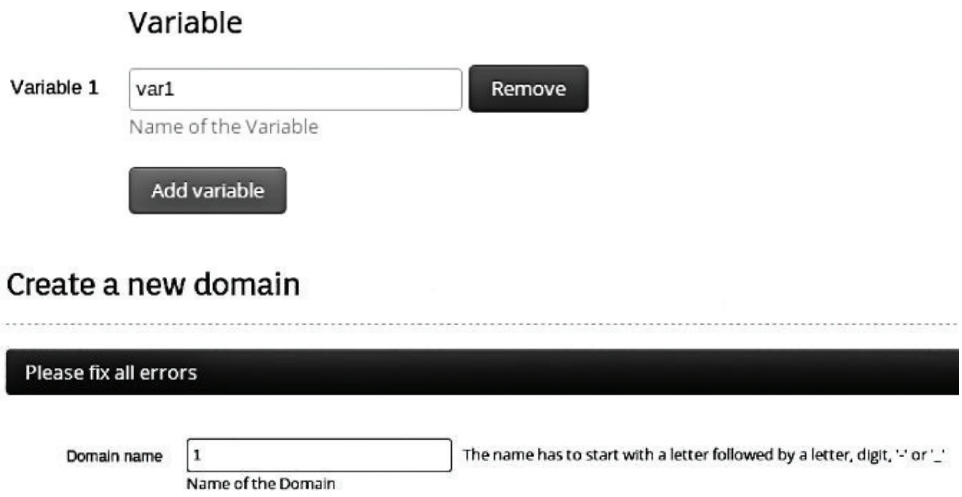


Figure 6.4: Example of creating a variable and creating/editing a domain with an error.

6.3.3 Implementation and services

The Planner is implemented in Scala entirely. To provide quick access to predicates in the state, an Internal argument container class keeps a map of (arity, index, argument) for the predicates that share same properties (the arity indicates the number of arguments in a predicate). This way, it is possible to look up the list of predicates where a specific argument is already provided, and to minimise the number of bindings that need to be checked when iterating over the possible bindings. We refer to a tuple that contains unbound variables as bindable, and to every possible substitution for unbound variables as binding. Bindable and Binding represent two case classes on the programmatic level.

Consider that task preconditions contain unbound variables. When evaluating these preconditions with respect to the current state, the values that are substituted for the unbound variables need to be somehow carried over to other parts of the precondition. In the case of compound tasks, the values need to be transferred also to the task's corresponding method. We achieve this by using Scala Symbols to represent variables. That is, we use the same symbol in multiple parts of a precondition, which means that all parts will reference the same symbol of the tuple that was first matched against it. For example, assume the state contains the following predicates: `(lamp l1)`, `(lamp l2)`, `(colour l1 red)`, and `(colour l1 yellow)`. If some part of the precondition is `(lamp 'name)`, the state has a match for $'name => l1$ and $'name => l2$. The `'name` symbol must be carried over to parts of the precondition

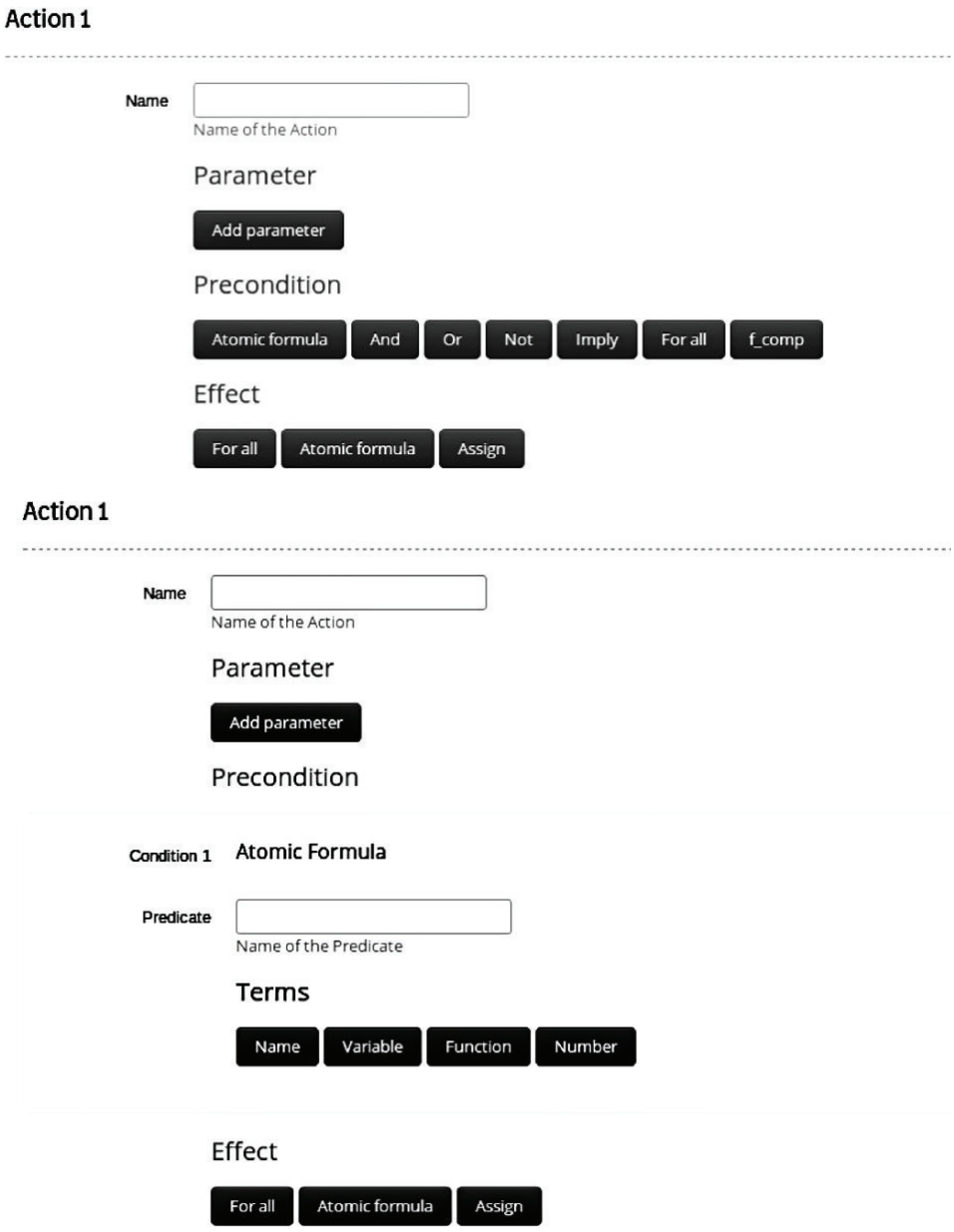


Figure 6.5: Example of guidance in creating a primitive task.

Table 6.2: Some of the services that **SH** provides.

Service	Description
Modelling services	
addDomain	Input: domain in string Output: acknowledgement
createProblem	Input: domain and problem names, requirements, objects, state, goal tasks Output: Problem
Problem-solving services	
planWithProvidedProblemInString	Input: problem, number of plans Output: set of plans
planWithGivenDomainAndProblemName	Input: domain and problem names, number of plans Output: print plans
planWithGivenDomainAndProblem	Input: domain and problem as objects, number of plans Output: set of plans
Management services	
checkDomainCorrectness	Input: domain in string Output: Success/Failure
Utility services	
checkSHLiveliness	Input: no Output: Success/Failure
convertPlanToJSON	Input: plan Output: plan in JSON

that are evaluated later. For instance, if a predicate for the colour of a lamp is added to this precondition and it becomes `(lamp 'name) and (colour 'name 'colour)`, the `'name` variable that is matched in the first part needs to be substituted in the second part. Then, the complete set of bindings for this precondition is $\{('name => l1,' colour => red), ('name => l2,' colour => yellow)\}$.

We provide planning as a service by implementing **SH**'s functionalities as Representational State Transfer (REST) resources (Richardson and Ruby 2007). Table 6.2 shows some of the services that **SH** provides. Upon receiving a request with appropriate arguments, **SH** may check the correctness of an HPDL domain or problem, the consistency of a required problem and domain, or search for a solution. The planning system replies to an interested component with an appropriate to a situation answer. For example, **SH** may provide a plan in JSON format, or it may show a syntax failure at a specific position in a domain encoding.

Table 6.3: Performance results in ms for **SH** and JSHOP2 ("OM" signifies "out of memory").

Domain	Problem (number of plans)	JSHOP2	SH
Blocks-world	1 (1)	44	162
Transitivity	1 (1)	/	123
Rover	1 (1)	5961	1617
	2 (10)	14165	2174
	3 (100)	OM	8351
	4 (1000)	OM	63659
DWR	1 (136)	98	546
	2 (1880)	1093	2844

6.3.4 Discussion

SH employs UTD (see Section 4.3.1) and it is therefore comparable to SHOP2 (or JSHOP2). JSHOP2, as a relatively recent and modern version, is characterised by complicated dependencies between its internal components which makes it hard to use and extend. Moreover, the planner uses code generation to transform HTN planning problems into executable code. On the other hand, we transform an HPDL planning problem into Scala objects directly, which enables convenient manipulation and extension. Moreover, **SH**, with small modifications, supports defining planning problems directly in code through the use of Scala Domain Specific Language. Both ways remove the code generation step of the JSHOP2 planner. Finally, with **SH** and its support for HPDL, we move closer to having a unified and well-defined language for state-based HTN planners (SIADEX also supports HPDL).

We execute a set of tests in order to provide insights into the performance of **SH** and JSHOP2. The tests have been run on an Intel Core 2 Duo @2.00GHz, 3GB RAM machine running Windows 7 and Java 1.6. The tests are based on planning problems generated for four different domains: blocks-world, transitivity, rover and dock-worker robots (DWR) domains. The first three domains are provided in the JSHOP2 distribution, while we model DWR for both planners as described in (Ghallab et al. 2004). For the blocks-world and transitivity domains, we require only one plan to be found. For the rover and DWR domains, we ask the planners to search for an increasing number of plans on the next problem. We run both planners on each HTN planning problem three times, and measure average values.

Figure 6.3 shows performance results for **SH** and JSHOP2. The second column includes the number of the planning problem and number of plans that the planners find within the showed time. **SH** shows worse performance than JSHOP2 in three domains, while in the Rover domain **SH** outperforms in terms of the planning time spent and the ability to find plans when JSHOP2 runs out of memory.

Chapter 7

Modelling and realising ubiquitous computing environments

So far, we dealt with two strings separately, one being the domain of ubiquitous computing and its issues when planning is in question; the other one is HTN planning, a technique that is desirable for solving problems in real-world domains and that we upgraded with features relevant to such domains. We saw that there are several attempts to tie these strings together in a knot (Chapter 2). Looking at the knot, we notice, however, that at least four important questions remain still open: What does a ubiquitous computing environment consist of and in what relation is it with an HTN planning problem? What happens with the plans computed for such problems? Finally, can this tie be realised in practice and what are the potential benefits of it?

Here we make a braid out of the two strings, aiming to answer these questions. In particular, we define a ubiquitous computing environment, including its physical constituents and their abstraction, activities that people perform, and a context which represents a snapshot of the ubiquitous computing environment. We also allow other kinds of information to be modelled, such as properties and conditions specific to the underlying environment. With these ingredients in hand, we form a ubiquitous computing problem and define its corresponding HTN planning problem. This in turn provides a good basis to discuss the soundness of this correspondence. Further, we use a pragmatic approach to deal with the dynamics of the ubiquitous computing environment during plan execution. When a dynamic event invalidates an executing plan, coordination either continues with the execution, if plan conditions allow for it, or it makes a new plan.

We designed, developed, and implemented our approach in a system prototype that is capable of full coordination in ubiquitous computing environments. We design our system following the SOC approach to obtain, among other benefits, scalability, evolution, and reuse of the system (Papazoglou and Georgakopoulos 2003, Erl 2007). We realise all components of the system, starting from devices that provide their functionalities as services, components that store environment data and process it, and those that deal with composition of device services using HTN

planning and coordination of plan execution and receipt of dynamic events.

We deployed and tested the system in the restaurant of the Bernoulliborg building with the aim to find out the benefits of automated coordination in ubiquitous computing environments. We evaluate the benefits of our system in terms of energy and monetary savings, satisfaction of the users with the system, and performance behaviour.

In the following, we first model a ubiquitous computing environment and detail how to get from such an environment to a ubiquitous-based HTN planning problem. Further, a description of the coordination at the execution level follows, which we refer to as orchestration. We then demonstrate how we realise the HTNPUC architecture introduced in Section 1.4 into a system prototype, and how we use the system in the actual environment. Finally, we show the results of several evaluations.

7.1 From environments to HTN planning

A formalised model of ubiquitous computing environments provides means for developing well-defined ubiquitous computing systems, which may bring many benefits, such as improved maintainability, ability to evolve, resuability, consistency, and sound reasoning over the context data (Bettini et al. 2010). Therefore, an approach employing planning for ubiquitous computing is well established when there is a clear understanding of how a ubiquitous computing environment is defined and how the corresponding planning problem is created.

7.1.1 Model of ubiquitous computing environments

Ubiquitous computing environments are enriched with assorted devices, which report various kinds of data used to interpret the environments, and some of which provide means to adjust the environment to meet users' needs and operate efficiently at the same time. We therefore say that a *physical model* of a ubiquitous computing environment consists of a set of devices D, where each device d has a sensing service s_d that returns the device's output. To exemplify this, let us revive Theodore and place him in the office building where he works. The building environment is equipped with various types of sensors. In Theodore's office, for example, there are two light sensors, one embedded in the window to sense the natural light level, and the other one placed above Theodore's desk to gather the indoor light level. The values of these sensors are returned through *getLux* services.

The raw data coming from devices in D is then a set of outputs $\{s_{d_1}, \ldots, s_{d_n}\}$. In actuality, this set presents pieces of data that has little meaning and may be conflicting. For example, if we assume that the two light sensors in Theodore's office are

in fact different hardware components, it may happen that they sense contradicting values. In order to ensure a consistent view of the light conditions of the environment and to get more meaningful information from the raw data, it is necessary to combine the outputs of devices. We use a *data fusion* function $df(\cdot)$ to relate devices' outputs to variables, which represent an abstraction of the physical model. That is, $df : \{s_{d_1}, \ldots, s_{d_n}\} \rightarrow V$, where V is a set of environment variables. Here we do not assume that the application of $df(\cdot)$ guarantees assignments of values to all variables in V, resulting in a *partially observable* environment. We refer to the abstraction of the physical model as an *environment*.

7.1 Definition (Environment). *An environment E is a pair $\langle V, L\rangle$, where V is a set of environment variables such that each v varies over a domain $\bar{D}^v$ and has a location $l_v \in L$, and L is a set of locations.*

With this definition, one or more unique variables can be associated with the office of Theodore. For example, *room564Lux1* and *room564Lux2* variables indicate the luminance level expressed in lux within the domain of, lets say, $[0, 1700]$.

When performing activities, people usually interact with the environment through its devices, causing changes in device outputs, for example, changes in values of a movement sensor. User activities can be therefore identified and recognised via environment variables whose values are assigned with respect to device outputs. In addition, activities tend to be regular and repetitive. In other words, we can derive that specific environment locations are associated also with specific activities. For example, Theodore's office is a location where he works with or without his computer (PC) at his desk, or a meeting room, where people typically give presentations or have discussions. We refer to this association as *activity area*: a logically defined space where some particular activity takes place involving one or more users (Curry 1996).

7.2 Definition (User activity). *A user activity ua is a tuple $\langle n, l\rangle$, where n is the name of the activity and $l \in L$ is the activity area.*

This means that $ua = \langle workingWithPC, room564\rangle$ is the activity of working with a PC in Theodore's office. In reality, there can be many activities happening in one location. So, beside working with a PC, Theodore may work something without involvement of a PC, he may be just present in the office, or he can be absent. The occurrence of such activities depends on the values of variables abstracting the environment. This means that we can recognise all activities taking place at all locations in an environment given the variables associated with each location.

7.3 Definition (Activity recognition). *Activity recognition is a function $ar : E \rightarrow Act$, where Act is a set of activities.*

Each *act* $\in$ *Act* is a user activity derived from the correlation of all $v \in V$ that are associated with the location of *act*. Activity recognition can be achieved by various techniques, *e.g.,* (Riboni and Bettini 2011, Nguyen et al. 2014). The set of recognised activities together with the abstraction of the physical model provide a snapshot of the environment at a particular point in the time. We refer to such snapshot as a *context*.

7.4 Definition (Context). *A context c is a tuple $\langle E, Act \rangle$, where $E = \langle V, L \rangle$ is the environment and Act is the set of recognised activities at all locations in L.*

7.1.2 Ubiquitous-based HTN planning problem

The context, through its variables, may not satisfy certain conditions of the environment related to the performing of the recognised activities. For example, if Theodore starts working with his PC, it may be that too much sun light is hitting his desk. Usually, for every living environment there is a set of quality conditions that an organisation or home must adhere to or maintain. Such conditions can be enforced by environment standards, corporate policies, health protocols, *etc.*, and can be related to activities with different concerns. For example, in an office building, many activities and therefore the performance of occupants depend on the quality of light in offices. The European standard for lighting in indoor work places defines that basic requirements, such as light level, should be taken into account for existing and future buildings in general situations and diverse particular activities taking place there (European Committee for Standardization 2011). Conditions may specify a recommended state (*e.g.,* the light level when working with a computer should be between 450 and 500 lux), or limit alterations (*e.g.,* up to a certain amount of carbon can be emitted). The environmental perspective of such conditions encompasses not only the social dimension, such as the quality of life of occupants and health of people, but also the economic dimension, including resource management.

The satisfaction of environment conditions depends on the ubiquitous computing environment, more specifically, on the variables and their values. We can abstract away an *environment condition* ψ as a propositional formula over $(v = val) \mid (v \neq val) \mid (v < val) \mid (v > val) \mid (v \leq val) \mid (v \geq val)$ such that $v \in V$ and $val \in \bar{D}^v$. For example, consider the recommendation for the light level in Theodore's office when he is working with the computer to be between 450 and 500. This means the following condition must be satisfied by the environment $(room564Lux1 > 450) \wedge (room564Lux1 < 500) \wedge (room564PC = active)$.

For these reasons, the set of recognised activities may require a change in the environment, a change that will support users in performing their activities according

to some prescribed conditions. This means that we need to check what has to be done in order to adapt the environment appropriately. Since the set of recognised activities implies such need, we consider the set as a *request* to be achieved.

7.5 Definition (Ubiquitous computing problem). *A* ubiquitous computing problem P^E *is a tuple* $\langle E, \Psi, Act \rangle$*, where* E *is the environment,* Ψ *is a set of environment conditions, and* Act *is a set of activities.*

Recall that our physical model consists of a set of devices. Beside sensors, some of the devices represent also actuators, which means they enable acting upon their states. We say that a device d, which is an actuator, has an *acting service* a_d that can change its state. For example, there are two actuators unobtrusively embedded in the lamps near by the door and windows of Theodore's office. These actuators are associated with an acting service that turns on lamps, and another services for turning off the lamps.

Given a set of acting services A, we say that α is a *satisfying adaptation* for P^E if and only if $\alpha \subseteq A$ is a sequence of acting services that transform E into one accomplishing *Act* according to Ψ. A request, *Act*, is *achievable* if and only if there exists at least one satisfying adaptation for it.

This gives sufficient information to illustrate how a ubiquitous-based HTN planning problem is formed. Assume P^E to be a ubiquitous computing problem and $\mathcal{P}$ an HTN planning problem as defined in Definition 5.2. Then, s_0 is the initial state *corresponding* to V, and consists of the following ingredients:

- A predicate p corresponding to a Boolean variable $v \in V$, only when v evaluates to true.
- A numerical variable $\bar{v}$ corresponding to a numerical variable $v \in V$ associated with a value $val \in \bar{D}^v$.
- A corresponding predicate for each property of the environment. An *environment property* is an expression of the form $prop(b, b')$, where *prop* is the property name, $b \in V$ and $b' \in L \cup T$ and T is a set of device types, or $b, b' \in L$.

The properties represent additional information about the environment, including spatial relationships, device typing, and variables' descriptions. We can achieve abstract spatial representation (see Spatial properties in Section 2.3.1) of the environment by defining interrelationships over environment locations. Considering Theodore's office, we can say that his *desk* is a sub-location of his *room*. We can also associate each variable with the location to which it belongs, that is, *room564Lamp1* is at *desk*. Further, each device representing an actuator may have an attribute that determines its type. For example, *room564Lamp1* is of type *ceilingLamp*. Finally, a

property may describe or relate a variable to its current value. For example, *light-level* relates the variable *room564Lux1* with its current value 570.

The next component, tn_0, is the initial task network corresponding to Act. We create the corresponding tn_0 if and only if there exist $t \in \mathcal{T}_n$ for all $act \in Act$ such that $t = n_{act}$ and $|\tau| = 1$.

The last component of an HTN planning problem we need to consider is the domain theory. In this context, we consider an acting service as a primitive task, where preconditions model only variables that the service deals with in order to let the service be executable, and effects model how the variables are changed after the service execution. This means we do not annotate the corresponding primitive tasks with environment-specific knowledge. We say that a primitive task o *corresponds* to $a_d \in A$ if and only if $t(\tau) = a_d$, $pre(o)$ is a logical expression over V, and $eff(o)$ is a conjunction of expressions over V.

Thanks to HTN planning, we can present all environment-specific knowledge in compound tasks. This knowledge includes the environment properties and environment conditions. Generally, the environment-specific knowledge can also contain preferences of users, and may be used to empower users with a set of tasks that can be executed along with the automated coordination of the environment. In all cases, the environment-specific knowledge can be organised in some form of *complex services*. For example, a service to adjust the light level in a room. This service may have requirements involving environment properties and environment conditions, and several types of reactions that include acting services (or other complex services). Then, a compound task *corresponds* to a complex service, where the task's methods define the ways in which the service can react, and preconditions correspond to the service's requirements. The acting services and other complex services involved in the reactions of the complex service correspond to the methods' task networks.

We can now propose a correct correspondence between an HTN planning problem and a ubiquitous computing problem.

7.6 Proposition. *Let P^E be a ubiquitous computing problem and $\mathcal{P}$ be an HTN planning problem. $\mathcal{P}$ corresponds to P^E if and only if*

- *s_0 is the initial state corresponding to V,*
- *tn_0 is the initial task network corresponding to Act,*
- *$\mathcal{V} \subseteq V$,*
- *$\mathcal{Q}$ is the set of predicates corresponding to Boolean variables in V and environment properties, and*

- *$\mathcal{T}$ is the set of tasks such that primitive tasks correspond to acting services in A and compound tasks correspond to activities in Act and complex services based on environment properties and Ψ.*

The following properties hold.

7.7 THEOREM. *Let P^E be a ubiquitous computing problem and $\mathcal{P}$ be the corresponding HTN planning problem. If a request Act is achievable, then there exist a plan π for $\mathcal{P}$.*

Proof. Assume that α is a satisfying adaptation for P^E such that *Act* is achievable. From Proposition 7.6, there exist an HTN planning problem $\mathcal{P}$ corresponding to P^E. Since *Act* is achievable for P^E, there exist a plan for $\mathcal{P}$ that accomplishes tn_0. □

We can now obtain that the solution of the ubiquitous-based HTN planning problem is an adaptation for the corresponding ubiquitous computing problem.

7.8 THEOREM. *Let P^E be a ubiquitous computing problem and $\mathcal{P}$ be the corresponding HTN planning problem. If there exist a plan π that is a solution to $\mathcal{P}$, then we can construct a sequence of acting services α based on π that is a satisfying adaptation for P^E.*

Proof. Assume that there exist a plan π for $\mathcal{P}$. Since $\mathcal{P}$ corresponds to P^E, we can map each primitive task from π to an acting service from A. Given that π is a solution to $\mathcal{P}$, it means that *Act* is achievable for P^E according to Theorem 7.7. Thus, the resulting sequence of acting services is a satisfying adaptation for P^E. □

7.2 Orchestration

We now have a correspondence between the problems of ubiquitous computing environments and HTN planning established. We still have to bring the dynamics of ubiquitous computing environments into perspective, that is, events happening continuously while a plan is in execution.

In this context, many approaches try to interleave planning with the execution of plans and context changes by continuously revising already computed plans so as to achieve the given goal (see Monitoring and recovery in Section 2.3.2). Although such continual planning may be a more or less suitable approach, depending on the nature of the adopted planning technique, it commonly causes performance deterioration to the underlying system. This is usually due to the time spent on revising a plan, and even more, ending up planning from scratch when the revision is unsuccessful. A simpler approach, *e.g.*, always planning when some context update or service failure is encountered, brings a trade-off between spending unknown time on revising and/or planning from scratch and accepting a computation time with a known upper bound – the one of the planning step.

Such a pragmatic approach still has some challenges to be overcome, such as dealing with continuous events due to the dynamics of the environment, consistent updates of the planning state, and continuous execution of plans for the respective HTN planning problems. We refer to the process of receiving events, creating HTN planning problems, and executing the corresponding plans as *orchestration*. Thus, the orchestration acquires the most recent information about the ubiquitous computing environment and listens to new events coming from it. The events, which indicate environment changes, are incorporated in the current planning state or task network. Upon the receipt of an event, the orchestration constructs an HTN planning problem and executes the computed plan, if such exists. While executing, the orchestration takes into account newly received events and responses of acting services. If plans with obsolete services are discovered, then these plans are modified only if the remaining services are independent from the obsolete ones. Otherwise, a new plan is computed.

7.2.1 Model

The problem we need to solve with orchestration is based on the state model defined in Definition 2.1. In addition, it involves events, which means that, beside the state-transition function, there is an update function that changes the values of variables with respect to assignments provided in the events. We assume that the assignments in events refer to different variables. The problem we deal with here is in fact a simplification of the one considered in (Kaldeli 2013).

7.9 Definition (Orchestration model). *Let E be an environment. An orchestration model Σ based on E is a tuple $\langle \mathcal{S}, A, \mathcal{E}, \gamma \rangle$, where*

- *$\mathcal{S}$ is a set of states,*
- *A is a set of acting services,*
- *$\mathcal{E}$ is a set of events. An event assigns a value val to a variable $v \in V$ such that val $\in D^v$,*
- *$\gamma : \mathcal{S} \times A \times \mathcal{E} \rightarrow \mathcal{P}(\mathcal{S})$ is a transition function $\gamma(s, a, \epsilon) = \{\delta(s', \epsilon) \mid s' = s[a]\}$, where $\delta : s \times \mathcal{E} \rightarrow \mathcal{S}$ is a function that updates a state s using the assignments from $\mathcal{E}$ and $s[a]$ is as defined in Section 2.1.*

The orchestration model is the one determining the orchestration. In other words, an orchestration o refers to a sequence of pairs of events and plans $\langle (\epsilon_0, \pi_0), \ldots, (\epsilon_k, \pi_k) \rangle$. The orchestration problem then concerns maintaining a consistent planning state given uncontrolled events, and finding and executing a sequence of acting services that satisfy some request given such state.

7.10 Definition (Orchestration problem). *An orchestration problem P^O is a triple $\langle \Sigma, V, Act \rangle$, where Σ is an orchestration model, V is a set of environment variables, and Act is a set of recognised activities.*

Upon each context change, a ubiquitous-based HTN planning problem is created with a state resulting from the application of δ. Thus, for a sequence of events $\epsilon_0, \ldots, \epsilon_k$, there is a sequence of ubiquitous-based HTN planning problems $\mathcal{P}_0, \ldots, \mathcal{P}_k$ and their corresponding plans $\pi_0, \ldots, \pi_k$. We refer to the sequence of plans $\langle \pi_0, \ldots, \pi_k \rangle$ as an orchestration plan Π. Since events may come fast in a sequence, many of these plans will contain redundant acting services. On the other hand, parts of some existing plans may be already executed before a new plan is set for execution and whose acting services may contain a subset of already executed ones. In both cases, we can reduce Π by removing the redundant acting services without introducing conflicts among dependent services. A reduced orchestration plan Π' is a sequence of modified plans $\langle \pi'_0, \ldots, \pi'_k \rangle$ such that it is a correct execution (see Definition 4.20), that is, it still achieves Act. Then, Π' is a solution to P^O if Π' is part of an orchestration o that produces a sequence of states such that achieves Act.

7.2.2 Algorithm

We design an algorithm that supports concurrent processing of incoming events and executing actions. In this way, the orchestration can process messages coming in parallel from the changes in the environment and service invocations, and will not get blocked by acting services unable to react on the first invocation. The algorithm is able to handle simple service failures and tolerate events that may affect the orchestration model. Thus, the algorithm takes a pragmatic attitude and refers to planning whenever an event is received.

The algorithm is shown in Algorithm 4. The receive function indicates that messages are received asynchronously. The message 'InitialiseEnvironment()' informs the algorithm that a ubiquitous computing environment has to be created. This happens only when the algorithm is booted up, and involves gathering all information relevant for the environment, such as variables, locations, device types, and properties. Given all these ingredients, the algorithm coordinates the creation of corresponding HTN planning constructs. Here we assume that domain theories are encoded in advance and available for the orchestration to retrieve given a domain name (ConstructDomain(*domainName*)). Then, the domain object is stored and available for all planning requests. The state is created dynamically with respect to V and environment properties (ConstructState()). When a planning request is received (PlanReq(*taskNetwork*)), the orchestration invokes **SH** given the current HTN planning problem $\mathcal{P}$.

Algorithm 4 Orchestrating incoming events and execution of corresponding plans

```
E, 𝒟, s, Π′ empty
tn ← default task
function RECEIVE
    case InitialiseEnvironment():
        Gather all information about the environment E
        Create corresponding planning constructs based on E
    case ContextChange(ϵ):
        if ϵ is an activity then
            Update tn
        else
            p ← CONVERTTOPREDICATE(ϵ.v,ϵ.val)
            UPDATESTATE(p)
        Plan for the latest tn and s
    case ConstructDomain(domainName):
        D ← CREATEDOMAIN(domainName)
    case ConstructState():
        s ← TOSTATE(E)
    case PlanReq(taskNetwork):
        P ← ⟨D, s, taskNetwork⟩
        π ←SEARCH(P)
        if π ≠ ∅ then
            Execute π
    case ExecutePlan(plan):
        A ← CONVERTTOSERVICES(plan)
        Add A to Π
        for a_d ← A do
            a_d completes
            case a_d is failure:
                Retry the service n times
                if a_d has failed n times then
                    Remove a_d from Π′
                    Plan for the same tn and current s
            case a_d is success:
                Remove a_d from Π′
                Check whether s is consistent with the effects of a_d
        end for
end function
```

The message 'ContextChange(ϵ)' refers to the events received from the environment. If ϵ represents a new value for a variable, either a predicate or numerical variable is updated in the state. Otherwise, the event is added to the network of tasks to be accomplished. Upon each event, irrespectively of its type, a planning request is issued to find a new adaptation for the environment.

The last message 'ExecutePlan(*plan*)' first maps the plan tasks into a set of acting services. If possible, some of these services are executed in parallel. When an acting service completes its execution, the orchestration checks the response. When a service fails, the orchestration re-executes it for a fixed number of times. If this step is unsuccessful, the failure is permanent (see subsection Action contingencies in Section 2.3.1) and the algorithm removes the service from the reduced orchestration plan and issues a planning request.

7.3 Implementation

With the orchestration, we complete a whole operating cycle of a ubiquitous computing environment, starting from the collection of data through sensors, processing it into context information, planning the coordination of acting services in order to adapt the environment according to its conditions, until the execution of acting services upon devices representing actuators. In order to see what the benefits of such an approach are and whether the quality of life is improved in terms of energy savings and user satisfaction, we implement our approach in a prototype system ready for deployment. In the following, we provide insights into the realisation of each component of the HTNPUC architecture, focusing on planning and orchestration ones.

Devices

We use wireless sensors that are based on the TelosB platform and compliant with IEEE 802.15.4, produced by Advantic Systems (*Advantic Sys.* 2015). Examples of sensors include natural-light sensors and motion detectors based on passive-infrared sensors. The sensor nodes are implemented in the nesC language and run on the TinyOS embedded operating system.[1]

Another type of sensors we use are plugs for measuring electricity of appliances, produced by Plugwise (*Plugwise* 2015). These are plug-in adapters that fit between an appliance and a power socket. The plugs form a wireless mesh network around a coordinator. The network, which is based on ZigBee,[2] communicates with a base station through a link provided by a receiver (in the form of a USB stick). At the

[1] `http://www.tinyos.net/`
[2] `http://www.zigbee.org/`

same time, we use the plugs as actuators – they enable control of the power flow, and thus, turning on/off of attached devices.

At the gateway, there are two types of services all implemented in Scala: one type to handle sensors, and another one for Plugwise devices. The sensor services handle readings in an asynchronous manner. Every time a new message arrives from the sensor network, a service checks the message senders and push new data to a publish and subscribe component implemented by RabbitMQ.[3] The services responsible for managing Plugwise devices enable gathering the current state of a specific plug, and changing the plug's state (*i.e.,* switch a plug on or off).

Context

The context component, which is implemented in Scala, consists of two parts, namely activity recognition and context processing. Activity recognition takes care of user-activity related data (*i.e.,* motion data) and recognising the presence of user in each area of the environment. The user presence in an area can be concluded from one or several PIR sensors, depending on the configuration of the environment. We also implement a dynamic presence detection mechanism that adapts the timeout of the presence activity given the time of the day. For example, in the early morning, the timeout is set as short as five minutes, while during the lunch time, the timeout is set to 30 minutes in order to minimise any possible inconvenience.

We adopt an ontology-based activity recognition approach. The ontologies are developed using Protégé,[4] a graphical tool for ontology development that simplifies design and testing. The ontological reasoning is performed using the HermiT inference engine,[5] and its application programming interfaces for the Java programming language. The recognition algorithm is developed in Java and implemented as an on-line recognition system. More details are presented in (Nguyen et al. 2014).

Context processing is responsible for ensuring a consistent view over the ambient environmental condition. For example, this includes a calibration of the raw sensor readings coming from light sensors in the unit of lux.

As soon as new activities and environment changes are detected and processed, the context component, through RabbitMQ, informs the interested parties about the updates.

[3] `https://www.rabbitmq.com/`
[4] `http://protege.stanford.edu/`
[5] `http://hermit-reasoner.com/`

Repository

We use two databases to store the information about a ubiquitous computing environment, each containing one type of information. Static information is stored in Neo4j,[6] which is an open-source database with features from both document and graph database systems. It promises features such as scalability, availability, and performance.

We store dynamic information in Cassandra,[7] which is a NoSQL database that delivers fast performance and is suitable for systems with time series or big data.

Domain model

We use the **SH** planner for computing compositions of device services. Details on **SH** are presented in Section 6.3. Here we demonstrate how to create a domain model specified in HPDL given a model of a ubiquitous computing environment.

We encode device types and locations as domain types, which are all subtypes of the type `object`. For example, consider that the restaurant in the building where Theodore works has two types of lamps: security and regular lamps. Then, `regularLamp` and `securityLamp` are domain types whose supertype is `lamp`, which is a subtype of `object`.

The properties of the environment are specified as predicates of the form `(prop arg1 - type1 arg2 - type2 ...)`. An example of a location-related property is `(in ?a - area ?r - room)`, which denotes that an area is within some room, lets say, the restaurant.

Boolean variables are encoded as predicates too. Such a predicate is in the state only when the corresponding variable evaluates to true. The predicate name indicates the truth value, and the predicate's only argument is the variable itself. An example of such a predicate is `(turned ?l - lamp)`.

Numeric variables are modelled as domain functions. A domain function returns the value of the variable it represents. The function name describes the sensor associated with the variable (we get this association from the databases). A function may also have arguments denoting variables to which the numeric variable is related. For example, `(light-level ?a - area)` encodes the variable representing a natural light sensor deployed within some area of the restaurant.

Acting services are encoded as primitive tasks or actions in HPDL terms. We use a parametrised action for the same type of acting services. Then, a parameter denotes the actual variable we want to act upon. In this way, we ensure the uniqueness of each action instance. An action may have other parameters that refer to the

[6] `http://neo4j.com/`

[7] `http://cassandra.apache.org/`

```
(:action turn-on-lamp
 :parameters (?l - lamp)
 :precondition (not (turned ?l))
 :effect (turned ?l))
```

Figure 7.1: Example of an acting service encoded in HPDL.

```
(:method deficient-light-level
 :precondition (and (>= (light-level ?a) ?x) (< (light-level ?a) ?y)
                    (in ?l ?y) (regularLamp ?l) (not (turned ?l)))
 :tasks (sequence (turn-on-lamp ?l) (estimate-light-increase ?l)
                  (increase-level ?a)))
```

Figure 7.2: Example of conditions contained in an environment standard encoded as an HPDL method.

input variables of the corresponding acting service. The preconditions and effects capture the semantics of the service. Since we have acting services without semantic annotations, our actions are very simple. The HPDL action in Figure 7.1 represents an acting service for turning on a lamp only when the lamp is turned off.

Complex services or domain-specific knowledge is modelled in compound tasks. Let us assume that we need to adjust the light level in some area within the restaurant. There are two possibilities for this: to increase or decrease the light level. These two possibilities indicate two complex services that require different modelling but similar reasoning. Thus, each complex service is a compound task, namely `increase-level` and `decrease-level` tasks. Each method encodes a specific way of accomplishing the respective task. Let us focus on the `increase-level` task. If there is not enough light, it means that we need to turn on some lamps. While doing that, we need to ensure a certain level of light according to, lets say, the European standard for lighting in indoor work places. Then, one method of the task deals with the case when there is deficient light and there are lamps that can be turned on. That is, if the estimated light level of the given area is not within the recommended light-level range, and there are lamps that can be controlled and are turned off, then we can turn on a lamp, estimate its effects on the light level (simulated sensing), and recursively call `increase-level`. The encoding in Figure 7.2 represents this case.
The process of estimating the effect of a lamp on the light level is encoded in a separate task. This is necessary because, first, we keep primitive tasks simple, which means they do not perform sensing, and second, we perform off-line planning with completely instantiated variables, and thus, deterministic outcomes of actions.

Another method deals with the case when all lamps are already turned on. To address all of the lamps, for example, in the area of our interest and in the areas near by our area, we use a *forall* expression. Figure 7.3 shows the expression that can be

```
(forall (?l - regularLamp) (and (or (in ?l ?a) (near-by ?l ?a))
                                        (turned ?l)))
```

Figure 7.3: Example of *forall* expression ensuring satisfaction of some conditions over devices of the same type.

```
(:task absence
 :parameters (?a - area)
(:method turn-off-all-lamps
  :precondition ()
  :tasks (sequence (turn-off-lamps ?a))))
```

Figure 7.4: Example of an activity encoded as an HPDL task.

used in a method of the `increase-level` task.

We model user activities as separate compound tasks. These tasks further involve other compound tasks to ensure a satisfactory state of the environment, resulting in a well-structured hierarchy of tasks. For example, consider the case of presence activity in some area. The corresponding compound task involves `increase-level` and `decrease-level` tasks, depending on the estimated level of light given the information in the current state. The task encoding the absence activity, which is demonstrated in Figure 7.4, is simpler and it indicates that we can turn off all lamps associated with the area of interest. However, this does not mean that the area's lamps cannot be turned on later during planning. If areas, which are near by this area, have presence activities and insufficient light level, then it means that some lamps of this area will be turned on despite its absence activity. This is taken into account in the hierarchy of the `increase-level` task.

The modelling of other activities can be realised using reasoning analogous to the one described so far.

Orchestrator

Since the orchestration algorithm is stateful (see Section 6.1), *i.e.,* it maintains a model of the environment, including the state, domain, task network and reduced orchestration plan; and, in order to support our design assumption (*i.e.,* concurrent use and updates of the state), we built the algorithm upon the Actor model (Hewitt et al. 1973). It receives messages asynchronously and reacts to them by making local decisions, creating other actors to handle specific and/or concurrent messages, sending new messages, and deciding what to do upon the next message received.

The orchestration algorithm is implemented in Scala. We refer to the implementation of the algorithm as *orchestrator*. Instead on concrete implementations, the orchestrator depends on abstractions of other components. All abstractions are

Table 7.1: Set of standard operations for manipulating a ubiquitous computing environment.

Operation	Input	Output
initialiseEnvironment	/	/
isEmpty	/	Boolean
addVariable	variable	Boolean
removeVariable	variable	Boolean
updateVariable	variable, value	Boolean
getEnvironmentName	/	name
getVariables	/	map of variables

founded upon the Cake pattern (Odersky and Zenger 2005).

In order to accept different types of environments, the orchestrator uses a trait called Environment, which is specified based on Definition 7.1. It includes a predefined set of operations listed in Table 7.1. In our case, the environment is represented by an implementation of an office building. The orchestrator populates a specific environment by retrieving the information from the repositories. The set of variables, their types, locations, and properties are gathered from a Static Repository, in our system represented by Neo4j. The initial values of variables are gathered from the Dynamic Repository, that is, Cassandra. Both repositories provide a unified set of operations. Then, the orchestrator subscribes to the Publisher, such as RabbitMQ, and awaits for messages, that is, events.

The orchestrator creates a domain object with the help of **SH** planning services, and uses the Problem converter to transform Environment in a planning state as described in Section 6.3.1. Upon each event, the orchestrator creates a corresponding HTN planning problem and invokes the core planning service of **SH**. When a plan is found, if such exists, the orchestrator translates the plan steps into acting services and uses the device services implemented as REST resources for execution.

The orchestrator itself is built as a container for the Docker platform,[8] which automates the process of deployment of distributed applications.

7.4 Evaluation

We have deployed the system in the Bernoulliborg building in order to assess the possible benefits of our approach. While here we demonstrate the experiments we made with our system in an actutal environment, a previous version of the system and outcomes of its testing in a semi-simulated setting are provided in (Georgievski et al. 2013). Here we also evaluate the opinion of occupants of the environment

[8] `https://www.docker.com/`

Figure 7.5: Overview of the restaurant from the east and west sides.

with respect to several usability factors. Finally, we provide some insights into the performance of **SH** given the domain model we demonstrated earlier.

The restaurant that we chose as an actual environment is shown in Figure 7.5. It covers a total area of 251,50 m^2 and has a capacity of 200 sitting places. The restaurant has glass walls from three sides, enabling a significant amount of natural light to come through when the weather conditions allow for it, of course. The restaurant area is used for lunch in the period from 11:30 a.m. until 2:00 p.m. Outside these hours, the area is used by staff, students or other visitors for working, meeting, or other social activities.

The restaurant area is an open space divided in two sections by construction. We make use of this division in our testing. The layout is illustrated together with the locations of deployed sensors and electricity plugs in Figure 7.6. In particular, each section has 15 controllable light fixtures (or lamps), making 30 in total. There are several light fixtures that are uncontrollable and represent security lamps. While we do not control these, we take into account the light that they provide. In addition, there are two types of controllable lamps. The first type are large lamps that have 38W of power consumption each, and the second one are small lamps, each of which has 18W. These lamps are controllable thanks to the actuators we installed on them, which also serve as sensors by providing information about the fixture's power consumption. We installed 15 more sensors, one to measure the natural light level, and the rest to detect people's movement. In order to make a more meaningful use of the restaurant space given the movement sensors, we divide each section into smaller spaces, called *areas*. The areas are not necessarily of the same size, and we embedded movement sensors in each area in positions that cover most of the space of the respective area.

We conducted tests on the system over the course of five weeks in the months of February, March, and April 2015, involving measurements from Monday to Sunday. In the last three weeks in February, we recorded measures of energy consumption of lamps in order to understand the typical behaviour of manual control

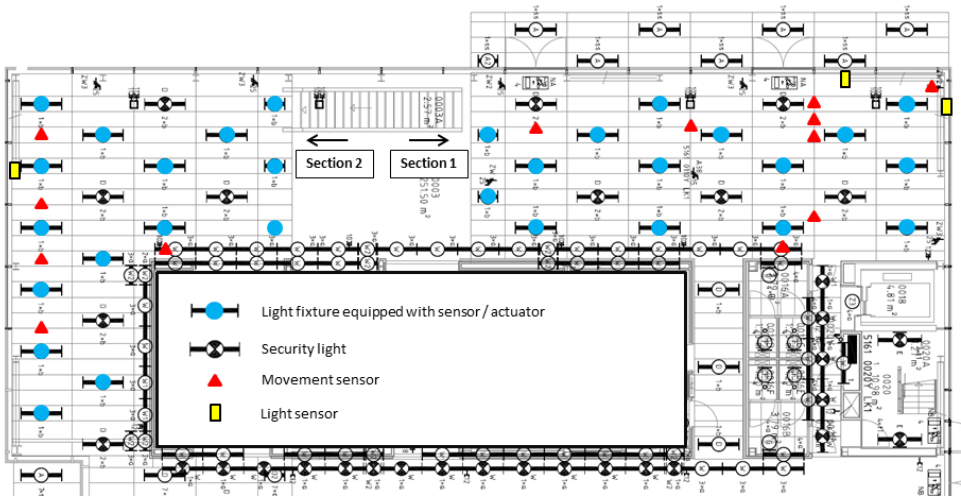


Figure 7.6: Schematic representation of the restaurant and deployed devices.

of lamps in the restaurant. This enables us to define a baseline. In the last week of March and first week of April, we allowed our system to control the environment in order to obtain the benefits of the system. Thus, manual control was disabled and the system was running continuously without interruptions during these two weeks.

The system offers interesting energy savings. These are due to the coordination of lamps given weather conditions, presence of people and a set of minimum requirements for satisfactory level of light, and monetary savings, which result naturally from the reduction of power consumption.

7.4.1 Energy savings

Observing the measurements gathered in February 2015, when the restaurant was controlled manually, we find that the average time point when the lamps are turned on is 6:30 a.m, and they usually stay turned on until 8 p.m. This means the average consumption per working day in the restaurant is 14 kWh. For weekends, there is no manual control of the lamps, thus no consumption.

The use of our system results in intelligent adaptations of the restaurant with respect to the natural light and presence of people. The implication is that there are a plenty of possibilities for lamps to be turned off. Thus, this provides for energy saving. Figure 7.7 shows the average energy consumption when the lamps in the restaurant are manually controlled and when our system is used. In contrast to the manual control, which assumes almost fixed time points for turning on/off

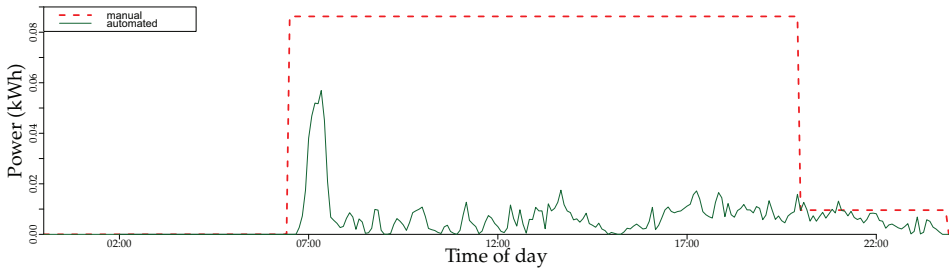


Figure 7.7: Comparison of average energy consumption between manual control and our system.

lamps, our system reduces the consumption of these lamps by turning them on only when really necessary. Figure 7.8 depicts this situation. The charts show the intelligent use of lamps and therefore energy in each day. The upper part refers to the first week of using our system, and the bottom one depicts the results for the second week. We also include the estimations of consumption if manual control would have been used. In addition, the figure includes weekends when there is no manual control provided regularly. Though the presence in the restaurant at evenings and during weekends is rare, there are still special occasions that our system encountered without any intervention (see Friday evening and Saturday on the upper part in Figure 7.8). This demonstrates that our system makes the restaurant truly adaptable to the happenings within it. To have a fair comparison, we assumed that in cases of special occasions, there would be manual control of the lamps provided in the restaurant. Also, one can notice that Friday in the second week is a special occasion too, that is, a holiday. In summary, the average savings of energy between the scenario of manual control and the one with our system is in the order of 80%.

7.4.2 Economic savings

We can also have an insight into the amount of money that needs to be paid for the periods of manual control versus our system. Of course, the proportion between the two cases is the same as with the energy consumption, and the price for an average day when our system is used is $0.37. Even in the worst case during working hours, which would happen when weather conditions are worse than on average and the restaurant is visited more than usual, the price resulting from the use of our system stays strictly within the boundaries of the one paid if lamps are manually controlled. Two situations, which do not occur in the case of manual control but happen when using our system, may imply leakage of money to be encountered during evenings in working days and on weekends. Assuming that environment conditions require a higher level of light, the situations involve turning

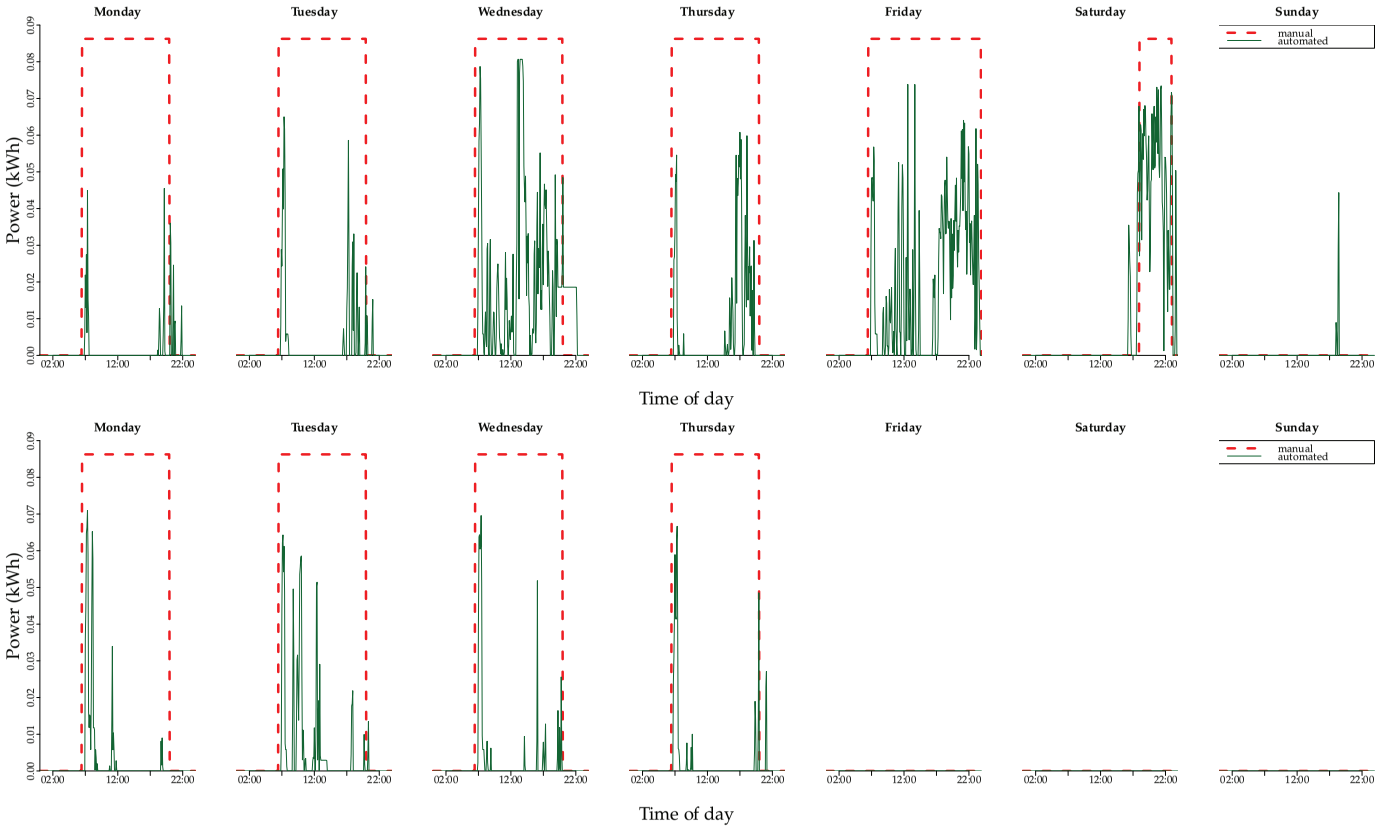


Figure 7.8: Comparison of energy consumption between manual control and our system for each day during week one (upper charts) and week two (bottom charts).

on lamps due to people passing through and not staying in the restaurant. It is difficult to extrapolate such situations from the data, but one obvious example is Sunday in week one. The price paid for this situation is negligible, or 0.78% from the amount of money we save in an average day.

In summary, considering a monthly bill paid for it when manual control is used, the use of the system implies economic savings such that allow for paying 7 months in total using the same amount of money as in the bill for one month of manual control.

7.4.3 Usability

The opinion of users regarding the use of our system in the restaurant is an important factor. We consider usability as defined in Section 2.3.3, and we prepared a questionnaire as a means to perform the usability testing. In order to have a well-formed questionnaire and focused evaluation, we use the guidelines for creating the evaluation from Section 2.3.3.

- Determine users, which involves identifying the set of possible user groups with their own specific goals and varying levels of effectiveness, efficiency and satisfaction. In our case, we focus on two groups of users, one experiencing the system during lunch, and another one outside lunchtime.
- Determine user goals, which involves the following aspects:
  - Acceptability, indicating the attitude of users towards our system. This includes the use of sensors, switching lamps more often than usual, automation of tasks, *etc.*
  - Learnability, referring to the need for users to understand how to use our system. For example, do users know how they can trigger the lamps?
  - System effectiveness, comprising the satisfaction of users with the overall system.
  - Efficiency, signifying the satisfaction of users with the time the system takes to perform its tasks. While this aspect is more technical, users can still evaluate how they perceive reactions of our system.
- Determine the context of use, which involves requirements of users based on the reason of "use" of the system. Since the system is unobtrusively integrated into the environment, there is no actual or intentional use of it. As for the requirements, for example, users having lunch may not have the same expectations for the level of light as compared to the ones reading or working in the restaurant outside lunchtime.

- Determine the level of importance, effectiveness, efficiency and satisfaction, which involves deciding the rating scale in an informed way as this defines the actual usability of the system. We use two Likert scales each with five levels, thus ensuring a symmetry of categories and therefore clarity when observing (Likert 1932). The format of the first scale includes the categories: totally disagree, disagree, neutral, agree and totally agree, while the second one offers: not useful at all, not useful neutral, useful, and very useful. Some questions have an additional category that captures the situation when users do not have an answer for or cannot answer a respective questions. This category can be either "I do not know" or "Does not apply".

In the end, our questionnaire has 24 questions, two with multiple choices and the rest with the items on the Likert scales.

Set-up

We conduct the survey on two groups of occupants, one experiencing the system during lunch (L), and another one outside lunchtime when working/reading (W).

We collected inputs for the questionnaire from group L on April 7 and 9, 2015, resulting in 54 entries in total. The majority of participants are visitors of the building (57%), while the others are occupants who are working (26%) or studying (17%) in the building. Most participants of group L use the restaurant for lunch only (96%), and the rest, beside using it for lunch, go there to study or work.

Group L consists of a high number of participants who believe that they are sustainability aware (83%), and those that engage in environmentally friendly behaviour (80%). Knowing that participants are savvy about how their actions affect the environment, we can expect with certainty that they appreciate a system for automated light control, and that they will not "use" the system unless it is really necessary. In this context, a high percentage of participants are familiar with automated control in buildings (65%), and the fact that lamps can be triggered by movement sensors (83%).

We collected inputs from group W on May 4 and 7, 2015. The total number of participants is 18. Most of the participants are students (72%) and the rest are visitors of the building. This group uses the restaurant for both working/studying and lunch (61%), some only for lunch (22%), 11% for playing games, and 6% for doing some business.

Group W also has a high number of participants believing that they are aware of sustainability issues (78%), and engaging in environmentally friendly behaviour (89%). More than half of the group is familiar with automated control in buildings (56%), and possibility that lamps are triggered through movement sensors (67%).

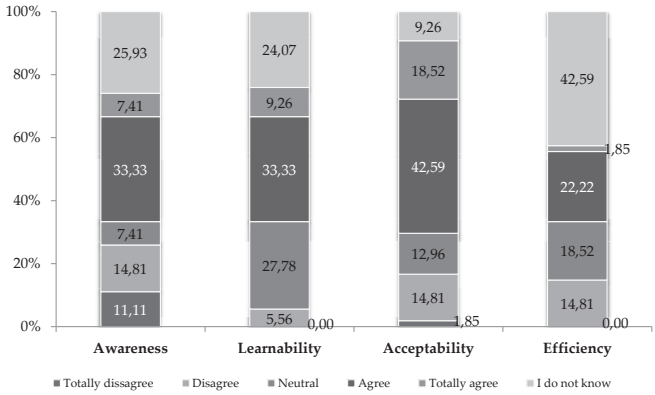


Figure 7.9: Results from the occupants of the restaurant during lunch with respect to several aspects.

Results

We organise the results of the survey on the basis of awareness, perception, acceptability, learnability, efficiency, effectiveness, and usefulness. In the questionnaire, each aspect is represented by one or more questions. In the following, we discuss the results of each aspect per user group.

Let us begin with group L whose results are shown in Figures 7.9 and 7.10. With respect to the awareness, one third of the participants stated that they are aware of our system, and one fourth is not aware, and another fourth did not answer the respective question. We then ask people what their perception about the purpose and capabilities of the system is. The majority of this group has a good understanding of what the system does. In particular, 65% know that the system is able to save energy and that it considers the natural light level (54%) and their presence (72%).

Though there is no explicit learning on how to use the system (learnability), 43% stated that it is easy for them to use the system in a sense to let a movement sensor know about their presence. 54% of the participants are neutral or do not know the answer to the respective question. Considering acceptability, 71% find the system to be good as it is, and 17% think that it causes distractions in terms of switching lamps too often, sensors not capturing them (the need to wave), *etc.*.

Looking at the efficiency of the system, the majority of participants do not know whether the system reacts to changes or how fast it reacts, or their answer is neutral (61%). Less than double of this percentage find the system efficient, while 15% think that the system does not react immediately. In contrast to this and considering system effectiveness, 59% are satisfied with the system, 31% are neutral, and only 6% dissatisfied.

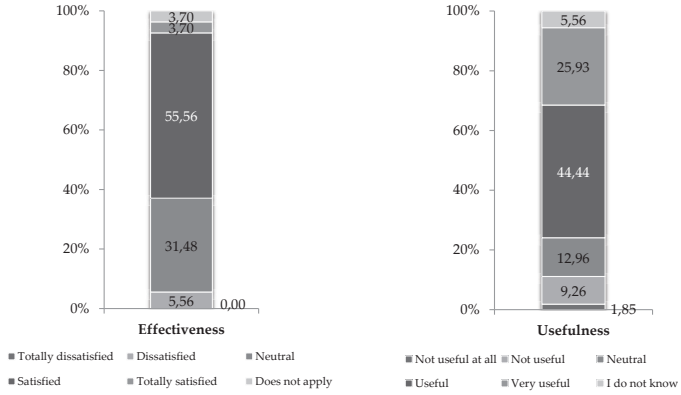


Figure 7.10: Results from the occupants of the restaurant during lunch with respect to the effectiveness and usefulness of the system.

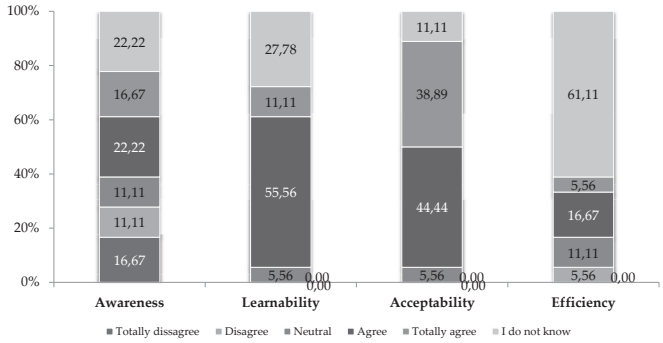


Figure 7.11: Results from the occupants of the restaurant outside lunchtime with respect to several aspects.

Finally, the majority of participants (70%) find the system to be useful, 13% are neutral regarding usefulness, and 11% are in opinion that the system is not useful.

The results of group W are illustrated in Figures 7.11 and 7.12. One can notice that the results for the awareness have similarity to those for group L, and indicate moderate awareness to our system. In particular, a bit more than one third of participants (39%) are aware of the system, 26% are not aware, and 26% do not have answer to the respective question. With respect to perception, the majority of the group know that the system is able to save energy (78%), while it takes natural light level (50%) and people's presence into account (67%). This shows that the participants of this group is well informed only with respect to energy efficiency and presence detection.

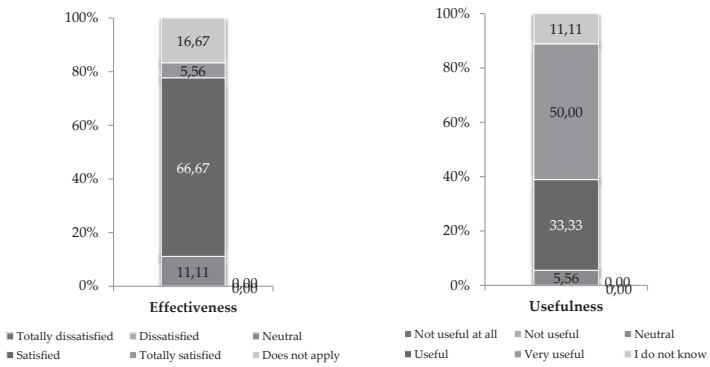


Figure 7.12: Results from the occupants of the restaurant outside lunchtime with respect to the effectiveness and usefulness of the system.

Regarding learnability, 66% of the group indicate that it is easy for them to learn how the system works, while 33.33% are neutral or do not know the answer to the respective question. Further, a large percentage of the participants accept the system (83%), while none of them think that it distracts them.

Considering the efficiency of the system, the outcome is similar to the one for group L. The majority do not know how the system reacts or give a neutral answer (72%). Around 20% find the system efficient, and 6% do not agree that the system is efficient. On the other hand, 72% of the participants are satisfied or very satisfied with the system, 11% do not express a feeling for this issue, and none of them is dissatisfied.

Finally, the majority of participants (83%) find the system to be useful, 6% are neutral, and 11% do not how useful is the system.

7.4.4 Performance

During the run of our system in reality, we recorded the HTN planning problem generated on each invocation of the planning system. The smallest planning problem consists of 177 state elements, while the biggest one of 207 state elements. The types of elements and their associated size in the case of the biggest HPDL problem description are shown in Table 7.2. The number of tasks in the initial task network is constant and equals to 13 (one task for each area). Then, the size of HTN planning problems encountered during the run varies between the ones of the smallest and biggest planning problem.

The number of invocations of **SH** per day is shown in Figure 7.13. The average number of HTN planning problems is 264, the average number per working day is

Table 7.2: Sets of state elements and their sizes in the biggest HTN planning problem encountered during the run of the system.

Elements	Size
Room variables	2
Area variables	13
Regular lamp variables	30
Security lamp variables	9
`light-level` functions	13
`in` predicates	52
`near-by` predicates	36
`turned` predicates	39
Initial task network	13
Total	207

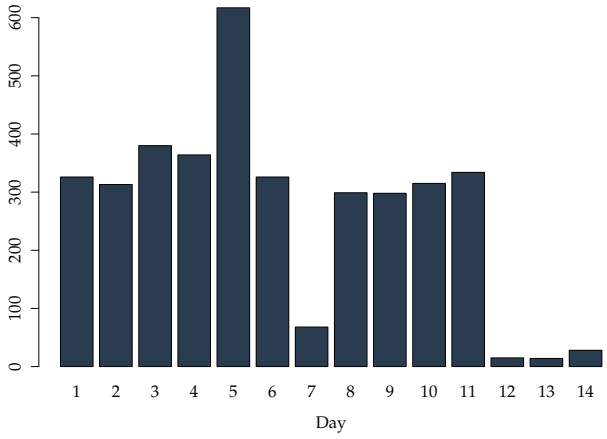


Figure 7.13: Number of invocations of **SH** during the two weeks of system run.

360, while the average number per weekends/holidays excluding exceptional situations is 31. In fact, two days are exceptions. One working day (Day 5) that has almost twice times more invocations than the average number of invocations in working days. The second one is a weekend day (Day 6) and has a number of invocations unusual for weekends. Both cases are a result of the special occasions in the restaurant.

Given the number of HTN planning problems that need to be solved per day, and that planning is computationally expensive task, we execute a set of perform-

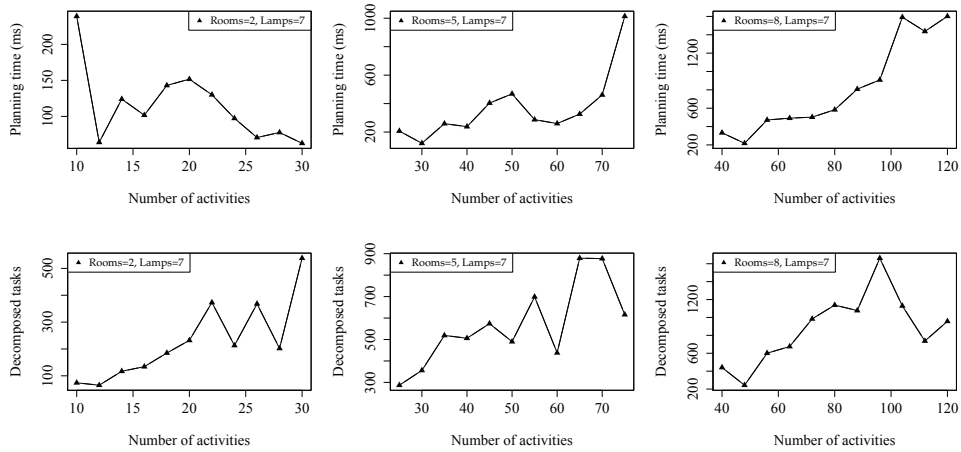


Figure 7.14: Performance results of **SH** when scaling the number of tasks in the initial task network (the number of lamps per area).

ance tests. We want to see whether the planning system can remain practically useful in larger and more complex environments. The tests have been run on a Intel Core i7-3517U @1.90GHz, 8GB RAM machine running Windows 8.1 and Java 1.8.0_31. The tests are based on the HPDL problem description created during the system run in the real environment, and use the same domain model. We run **SH** on each HTN planning problem three times, and measure and present mean values.

We generated two sets of HTN planning problems by changing their load profiles. In the first set of tests, we evaluate the performance in terms of scalability of the number of tasks in the initial task network under a constant number of lamps per area. We are interested in the behaviour of **SH** when the size of the restaurant increases. Figure 7.14 shows the scaling of **SH**. While the number of lamps per area is fixed, the uppers charts indicate the planning time and the lower charts depict the number of tasks decomposed both in function of the number of tasks in the initial task network. In addition, moving from left to right charts, we increase the number of rooms too. Looking at the upper part in Figure 7.14, it is difficult to assess the behaviour of the planning system from the leftmost chart. From the other two charts, one can notice more or less a gradual rise of the planning time up until 70 activities in the middle chart, and until 100 activities in the rightmost chart where the number of rooms is higher. After these points, the planning time increases steeply where the worst case is just above 1.5 seconds.

The lower part in Figure 7.14 demonstrates that the number of decomposed tasks increases on a regular basis, though one can still notice zigzag curves.

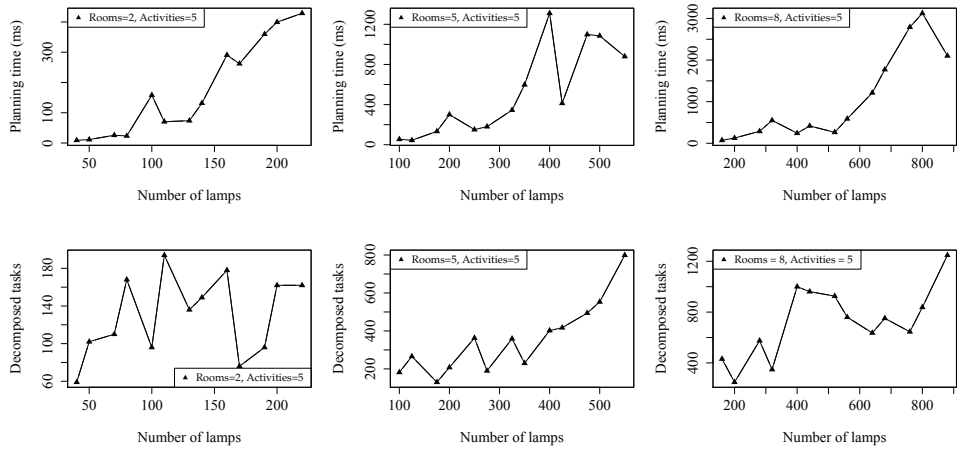


Figure 7.15: Performance results of **SH** when scaling the number of lamps per area.

In the second set of tests we evaluate the performance in terms of scalability of the number of lamps under a constant number of areas and therefore activities. Considering that lamps can be exclusively in one are or shared with other areas through the `near-by` predicate, this increases the difficulty of solving a planning problem. Thus, the number of predicates related to location properties increases too. Figure 7.15 depicts the results. The organisation of the charts is the same as in the previous figure. With respect to the planning time, one can notice that the number of lamps and their relationships with the environment layout affect the performance of **SH**. When the number of lamps is 200, the planning time is almost the same in all three charts despite the fact that the number of rooms is increased. The curves are gradually rising, and **SH** needs around 3 seconds to deal with 800 lamps.

In comparison with the lower part in Figure 7.14, the lower charts in Figure 7.15 do not present easily identifiable patterns. The middle chart depicts an oscillating curve up until 400 lamps, when the number of decomposed tasks increases fast. In the case of the rightmost chart, one extreme is at 400 lamps, when the number of decomposed tasks drop gradually, and then a rising curve to the other extreme for 900 lamps.

In summary, given the domain we modelled and simulated problem specifications, **SH** needs around 1.5 seconds and 3 seconds to deal with 120 tasks in the initial task network and 800 lamps, respectively. With respect to the number of decomposed tasks, there is no a clear correlation with the increasing factor.

7.4.5 Remarks

We remark that our system too consumes energy for its operation. It consists of 30 Plugwise devices, 14 sensors, one thin client, and one server that consume 3.3 W, 6.7 W, 4 W, and 365 W, respectively. The amount of energy consumed by the plugs and sensors represents a negligible fraction of the total consumption. While the thin client is an additional component we had to consider, the server represents a computer already in use, which adds little to its current consumption. As for the thin client, we believe that its consumption can be paid back in a very short period of time considering the savings obtained from average days of energy consumptions.

The Smart Meter and Scheduler presented in this chapter are contributions by Giuliano Andrea Pagani and Viktoriya Degeler, respectively.

Chapter 8

Coordinating cost-aware offices

The awareness of ubiquitous computing environments to energy and monetary costs is becoming a requirement rather than a coincidence. This means that in addition to automating the coordination of devices embedded into the surrounding, the environments must incorporate another dimension, the one of price and amount of energy used for such coordinated devices. This is especially true for the environments that are or will be connected to the smart grid. This in fact is our main motivation to bring another string into perspective, one of a different nature. That is, we still deal with coordination in ubiquitous computing environments, but we slightly deviate from planning and explore the benefits of an alternative approach to it.

The *smart grid* as a power grid provides an infrastructure for two-way communication between providers and consumers, digital metering through smart meters, inclusion of renewables, and dynamic pricing enabled by competing providers. A ubiquitous computing environment can take advantage of these possibilities by adapting its energy consumption to the price and availability of energy. In particular, the possibility of having real-time pricing and to be equipped with renewable energy generators will change the way ubiquitous computing environments are controlled. The first possibility is in line with the current trend in most countries, where the single provider/single tariff system has been replaced with models with competing providers and, basically, two prices over long-term contracts (usually in the term of months). The second possibility, that is, renewables are and will be increasingly present not only in a medium-large scale on the grid, but are also increasingly available at the level of a single ubiquitous computing environment, lets say, a building, as solar panels, wind and combined heat-power generators, and batteries. The building has to be aware of the energy generated locally in order to decide the proper policies to adopt: either use the energy produced for its local needs or feed the energy into the power grid and receive a payment for it. Therefore, the intelligent elements inside the building have to be able to know the energy produced on-site in order to eventually adapt their operations.

Here we present an approach to controlling an office building to save energy

and overall energy bill costs; this assumes the availability of a smart grid that offers dynamic prices from competing providers. The approach is based on (1) monitoring the energy consumption at the device level, (2) monitoring energy production of small-scale generating units, (3) associating policies for the devices that conform with user requirements for comfort and productivity, (4) controlling in an optimal way the energy consumption patterns of devices following the usage policies, and (5) being able to acquire dynamically the prices of energy from different providers and closing contracts for short-term time intervals.

For our approach, we design a system architecture consisting of several software components responsible for the monitoring of devices; storing data about devices, their policies and states, and energy providers; communicating with the smart grid; and scheduling and controlling the devices. We coordinate the execution of these components in a centralised manner. We have implemented the system in our own offices at Bernoulliborg and tested over a short period as a proof of concept. This initial investigation shows that automatic control of devices can reduce the overall energy consumption and, if coupled with dynamic pricing from the smart grid, can provide considerable financial savings from the end user perspective; this considers both the case of a building equipped with renewable-based small scale energy sources and the case without such installation that provides a baseline for the study. We do not investigate the provider's point of view, but we conjecture that also the provider will experience significant financial benefit if most of the end users would be price driven in their energy use as following prices means supporting easier demand/response, thus avoiding expensive energy peaks.

What follows is a presentation of the approach, including a model and general architecture. We then specify the technologies we use to implement our approach in an actual system, We also describe the living lab where the system was deployed, and show the benefits of its use on a set of experiments.

8.1 Approach

Among ubiquitous computing environments, we here focus on office buildings. The approach we take to save energy in buildings is based on a likely future evolution of the smart grid and on the possibility of associating policies with energy consuming devices.

8.1.1 Model

We assume that each building (or part of a building) is equipped with an interface with the smart grid that offers information on the price of energy proposed by different providers per time interval and possible maximum amount available

at that price. The time intervals are discrete and last one hour. Thus, contracts are electronically signed on an hourly basis, as each hour the price and amounts can be different.

From the point of view of the office devices, we assume that any energy consuming apparatus, *e.g.*, heater, fridge, printer, projector, can be measured in its electrical energy consumption in kWh and can be controlled. Each device has an associated state machine and an energy consumption level for each state. For example, a fridge consumes about 10^{-3} kWh when idle, but about 0.63 kWh when actively cooling. The system has full access to reading the state of a device and can trigger a state transition. Data about energy consumption levels are obtained by analysis of historical data for that type of device.

To avoid changing states of devices too often, we propose the notion of the minimum time unit. The minimum time unit is an adjustable parameter that tells the system how rapidly the devices can be forced to change states.

For each device, there is an associated *policy*. A policy is a set of consistent rules that hold for device operations. For example, "a fridge must work at least 15 minutes per hour" to be able to maintain its internal temperature below a certain threshold temperature level. Policies can have different parameters, a few of which are common to all: $(tBegin, tEnd)$ – time period, when the policy is active; and sid – state ID that the policy is applied to. State IDs are unique per device. In general, we assume several possible states per device, together with associated actions to move a device to these states. In the presented setting, each device has two states: "on" and "off," and two associated actions: "turn on" and "turn off."

We define and use five types of policies, which represent common rules for widely deployed devices. The five policies are summarised in Table 8.1 and defined next.

REPEAT. The device must be operated cyclically by entering the state sid repeatedly with a certain periodicity. For example, a fridge that should operate for 15 minutes each hour is specified using this policy. Parameters specific to this policy are: $tCycle$ – a total cycle time; and tOn – a time during this cycle, when the device should be in a state sid.

TOTAL. Specifies a total amount of time tOn that a device should be put in a state sid. An example is a laptop that needs recharging for two hours; however the exact time when it is going to happen does not matter, as long as it stays within $(tBegin, tEnd)$ bounds. This policy also assumes that the time when a device is in the state sid can be split into several parts. For example, we can charge a laptop for half an hour, then for another hour a little later, and for another half an hour even later.

Table 8.1: Device policies.

Policy type	Associated device	Description
REPEAT	Fridge, Boiler	Device should be put to a specified state repeatedly with a certain periodicity.
TOTAL	Laptop	Device should operate for at least a certain amount of time.
MULTIPLE	Printer	Device should operate for the time that allows for all scheduled jobs to be performed.
STRICT	Projector	A strict schedule is given in advance.
PATTERN	Microwave	An expected pattern of device operations.
SLEEP	Any device	No demand for device during the scheduling period.

MULTIPLE. Devices that schedule a number of jobs over a certain period of time use the $MULTIPLE$ policy. It has two specific parameters: $nJobs$ – a total number of jobs to be scheduled; and $tDuration$ – a time needed to complete a single job. An example is a printer that processes large batch jobs (*e.g.*, printing a book): each job needs 15 minutes to be completed, and a total of three jobs are required to be performed. With such a policy it does not matter when a particular job is scheduled, but it is important that the device is not turned off in the middle of performing a job.

STRICT. To enforce a state sid to be active from $tBegin$ to time $tEnd$, the $STRICT$ policy is used. An example is a projector that should be turned on at the beginning of a meeting and turned off when a meeting ends. The policy firmly defines the schedule for this device, as times are strict, so the system has no possibility to change the energy consumption time of the device.

PATTERN. The $PATTERN$ policy provides information about a way the device consumes energy. Instead of offering the possibility of controlling the device, it provides information on expected energy usage that can help to schedule other devices. For example, while a microwave is never completely turned off, the energy consumption in stand by mode is much lower than the energy consumption when it is actively in use. Historical data show that a higher level of energy consumption is expected during lunchtime, so the system takes this into account when scheduling other devices.

SLEEP. For a device for which there is no demand for operation during a given period, the $SLEEP$ policy can be used. The policy is used mostly at night, when there is no activity in the office and many devices can be turned off in order to save

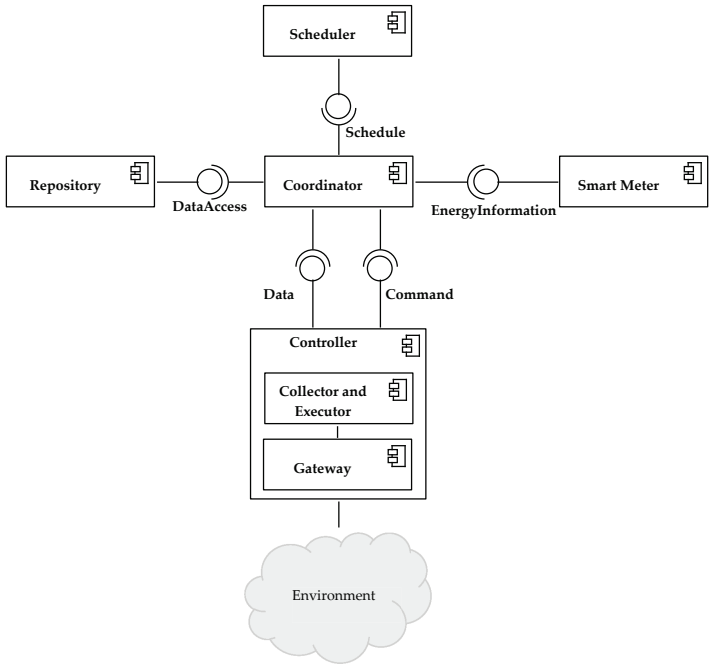


Figure 8.1: Component diagram of the architecture specified.

energy. There are no additional parameters for this policy.

8.1.2 Architecture

To take advantage of the dynamic pricing on the smart grid and the controllability of the devices, we design an architecture that goes from the hardware level of energy measurement and control up to the scheduling logic. The overall architecture is shown in Figure 8.1. On the right is the Smart Meter, intended as the interface to the smart grid and responsible for the two-way communication. At the bottom sits the hardware responsible for the monitoring and control of energy use (Environment), above which there is the Controller acting as a bridge between the Coordinator and the hardware. On the left side, the Repository contains historical data of energy use and the policies for the devices. This information is essential for the Scheduler (on top), who needs to plan, based also on the information from the smart grid, optimal control strategies for the office. The Coordinator component at the centre of the figure acts as a facilitator between the devices, the smart grid, the Scheduler, and the Repository.

Smart Meter

A smart meter is a physical device that is able to measure consumed and produced energy, provide this information to the energy metering companies, and change electricity tariffs according to the signals received by the energy companies participating in the smart grid real-time tariff service. In the proposed architecture, the Smart Meter is seen as the component that interacts with the smart grid in order to receive the energy prices that are applied by the different energy providers for the same hourly time interval. We envision a service, either from the smart grid itself or from energy providers, to provide through the Internet the changing energy prices. Once the information is received or retrieved, the Smart Meter component stores the data in the Repository through the interaction with the Coordinator component.

Once the generation and consumption data are available, it is then easy for the Smart Meter to provide this information either periodically or upon request to the energy provider for accounting/billing purposes. In the current implementation, we consider one smart meter in the office environment. However, the proposed architecture supports, with minimal changes, more smart meters, for instance, a smart meter per office floor, or per section of the building, or even per working business unit.

Environment

The Environment, most usually realised as a Wireless Sensor Network, provides the basic infrastructure for gathering the information on a device's power consumption, the device's state, and controlling appliances. Typically, this type of energy monitoring equipment is plugged into power sockets instead of running on battery. In addition, it has embedded wireless chips that are sufficient to form a wireless mesh network around the Gateway, providing a cost effective and dynamic high-bandwidth network, with a relatively stable topology. One can also envision these functionalities to be directly available at the appliance level (*e.g.*, a laptop that offers external control and energy consumption values as system calls that can be remotely invoked (Kremer et al. 2003)).

Controller

The Controller consists of a Collector and Executor subcomponent and a Gateway between the Environment and the above components. The Gateway is in charge of managing the network. It runs in the background, providing basic tools to the Collector and Executor for gathering information as well as controlling the devices. The Collector and Executor subcomponent, in turn, is responsible for the collection and storage of the office information. On a regular basis, the Collector

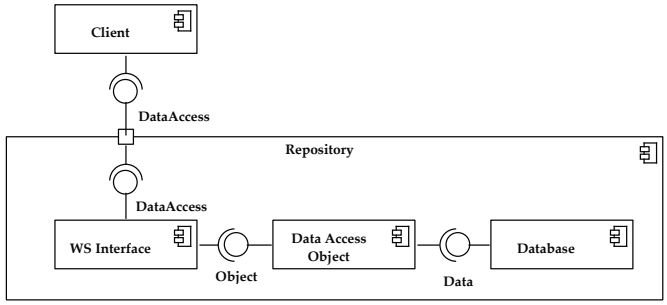


Figure 8.2: Overview of the Repository component.

collects the devices' data gathered through the Gateway subcomponent. In order to access lower-level tools of the Gateway in a more intuitive fashion, the Collector and Executor subcomponent contains a wrapper that provides a standard interface for interaction. The information received is then stored into a database.

Another responsibility of the Collector and Executor subcomponent is the execution of the actions over devices. It uses its wrapper to interact with the Gateway in order to send the execution commands to the Environment.

Repository

The Repository component comprises two basic functionalities: (i) storage of information provided by devices, policy manager and energy providers; and (ii) retrieval of data queries issued by other components, namely by the Coordinator. Communication between the Repository and Coordinator is enabled by exchanging an agreed format of messages. In Figure 8.2, we schematise the internal architecture: its configuration is abstracted into three subcomponents: Web Service (WS) Interface, Data Access Object, and Database.

The *WS Interface* is a thin layer over the Repository that offers its capabilities across the network in form of Web services. By designing such an interface, we simplify the overall system architecture and the visibility of interactions is improved. We view each Web service as a resource on which a set of actions can be performed. Furthermore, such an action is mapped onto an operation of the lower-level *Data Access Object* component. Data Access Object encapsulates and implements all of the functionalities required to work with the data source. It persists the requests and information provided by the client calls into the *Database*. Naturally, the back-end database can be freely chosen.

Scheduler

Through the Coordinator, the Scheduler receives the information about the available supply and price of energy, and also the information about controllable devices, their levels of energy consumption, and their policies (rules of operation). Given this information, the Scheduler then finds the optimal solution with the minimum price paid for the total energy consumed over a certain period of time.

Prices on the market change regularly, say each hour, so the Scheduler takes into account varying prices over the course of the day and tries to schedule devices to operate at times, when the price per consumed kWh is the lowest. Generally, those prices vary from provider to provider, and the system can choose a provider to buy energy from. However, since providers have a finite energy supply, if many devices are scheduled to operate at the same time, their total energy consumption will likely be bigger than the cheapest energy supplier is ready to provide. That will lead to the necessity to buy energy from a more expensive energy provider.

Let $EP(t) = \{ep_i\}$ denote a set of energy providers at the time unit t, where each *energy provider* is represented by a tuple $ep_i = \langle cost, energy \rangle$, *cost* is the cost of 1 kWh of energy, and *energy* is the maximum amount of energy that the current provider can provide at the time unit t.

To calculate the accumulated cost that an office building needs to pay for the energy it consumes in a certain time unit, we need to sort energy providers by their price. Since we assume that a smart meter can choose which provider to buy energy from, it first buys energy from the cheapest providers, and then continues to more expensive providers, if the amount of energy the building needs to consume is bigger than the amount offered by the cheapest energy providers.

Thus the total cost that the building pays at time unit t if it needs to consume an amount of energy e is

$$cost(t, e) = min(\sum_{i=1}^{|EP(t)|} (k_i * ep_i.energy * ep_i.cost))$$

such that

$$\sum_{i=1}^{|EP(t)|} (k_i * ep_i.energy) = e,$$

where k_i is the coefficient that shows a fraction of energy bought from energy provider ep_i. In practice, k_i will be equal to 1 for the cheapest providers, then be in a region $[0, 1]$ for one of the other providers, and be equal to 0 for all more expensive providers.

An example of cost calculation for the energy providers in Table 8.2 is shown in Figure 8.3. For the consumption level of 2.1 kWh, the office building has to use

Table 8.2: Example of energy providers and prices.

Provider	Energy supply	Price per kWh
Internal Wind Turbine	0.214292	0.0
Internal Solar Panel	0.302314	0.122916
COMED	2.755946	0.282973
ATSI	3.154828	0.357123
AEP	2.411659	0.360658
more providers …		

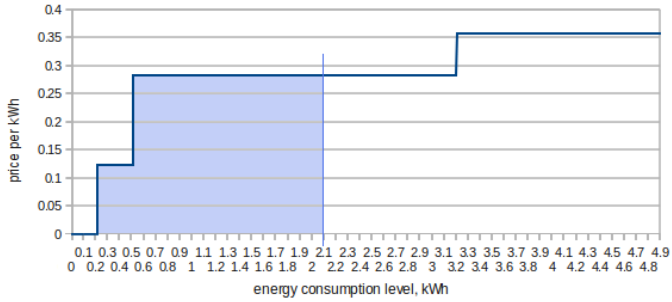


Figure 8.3: Price per kWh given the energy consumption. Total price paid equals the area under the graph.

energy from internal Wind Turbine and Solar Panels, and also buy some energy from the cheapest provider COMED, resulting in a total of $0.485217 per hour.

8.2 Coordination

The underlying mechanism of the Coordinator component is shown in Algorithm 5. The algorithm enables a coherent execution of the system as a whole. On a regular basis, the Coordinator asks the Smart Meter to provide the energy price information and sends gathered data to the Repository (lines 1-4). The Coordinator serves as a client to the Controller. Once the Collector and Executor subcomponent collects the device information, the Coordinator instance calls a specific Web service to retrieve that description, and, in consequence, it sends the data to be stored into the Repository to be available for later usage (line 6). The Coordinator also serves as a client to the policy manager to provide the system with policies needed by the Scheduler (line 8). At a point when all necessary input parameters for the Scheduler are secured, the Coordinator continues with the system execution flow by instantiating the Scheduler component (line 10). The received schedule of actions is controlled by this component too. Each action is scheduled for one-time execution

Algorithm 5 Execution coordination

Input: x: time period
1: **loop**{once per day}
2: $ei \leftarrow$InvokeSmartMeter($currentDate$)
3: Set the information from energy providers ei in the Repository
4: **end loop**
5: **loop**{every x hours}
6: $currentStates \leftarrow$InvokeController()
7: Set the current states of devices $currentStates$ in the Repository
8: $devices \leftarrow$InvokeRepository()
9: $prices \leftarrow$InvokeRepository()
10: $schedule \leftarrow$InvokeScheduler($devices, prices$)
11: **for** each action a in $schedule$ **do**
12: ScheduleAction($a.hour$,$a.minutes$,$a.mac$,$a.state$)
13: **end for**
14: **end loop**

by invoking the Collector and Executor component Web service to process changes deeper into the physical layer (lines 11-13).

8.3 Implementation

We have implemented the proposed system in a prototype that we have deployed in our own offices. Next we detail the realization of each component.

Interfacing with the smart grid

The smart grid has not yet been deployed and implemented for the end user, but has been used just as proof-of-concepts (Lu et al. 2010), simulations of smart grid customer behaviour (Taqqali and Abdulaziz 2010), or small scale pilot projects (Kok et al. 2008), and no generally available standards have been agreed yet (though initiatives are underway from IEEE, NIST, and others); therefore, we simulate the dynamic pricing.

To make the simulation realistic, we use data and services obtained from real markets and real energy generation installations. In particular, in order to simulate the variable energy tariffs, we use the energy prices coming from the PJM Interconnection[1], which is a regional transmission organization that coordinates the

[1] http://www.pjm.com/

movement of wholesale electricity in more than 13 states of Eastern U.S.A. The data extracted are the Day-Ahead Energy Market locational marginal pricing, which are the prices of energy negotiated in the wholesale market for the following day by energy companies at a specific location where energy is delivered or received. Data contain the energy price for each energy unit ($ per MWh) for each hour of the day agreed for the next day at 20 locations of delivery. We stipulate a maximum theoretical power consumption for our office building of little more than 4.2 kW; we assume that each simulated energy provider can provide in an hour a quantity of energy that is equal to a random value between 0 and 4.2 kWh. It is not then granted that just one provider can satisfy the energy needs of the office building, but more of them could be considered as energy providers at the same time.

Moreover, we consider the inclusion of micro-generation facilities as if they were available on the building. We simulate the presence of a photovoltaic (PV) installation and a small-scale wind turbine. Again, to make the simulation realistic, we use actual data coming from existing installations, a PV installation of 2.4 kW of power in New York at Dalton School in Manhattan.[2]

We simulate the presence of a small-scale wind turbine on top of the same building considering the average annual wind speed experienced in New York and the anemometer data obtained from the set of sensors measuring the environmental conditions on top of Dalton School. We simulate the presence on site of a Proven 2.5 wind turbine[3] which has a rating of 2.5 kW with a 12 m/s wind speed. We assume to have the wind data every hour and constant during the whole hour.

Regarding the pricing of the energy produced locally, firstly, we consider the wind turbine as a *sunk cost*, that is, the energy produced is for free, as its investment has been already amortized. On the other hand, for the PV we assume a price of $0.12 per kWh by considering the investment cost and the energy produced over the expected lifetime of the PV array. Secondly, we estimate a production of energy during the 40 years that is on average the same as the one produced in the previous years since the installation. Thirdly, the investment cost is based on the results of Wiser et al. (2009), who investigated the cost of PV panels in the U.S.A. The cost that emerges from their analysis, considering the cost for PV panels, inverters, and installation once the incentives applied by the U.S. government are subtracted, is $5.1 for each installed watt of power.

[2] `http://www.dalton.org/`

[3] `http://www.windandsun.co.uk/Wind/wind_proven.htm`

Environment

We use Plugwise adapters consisting of plug-in adapters that fit between a device and the power socket (*Plugwise* 2015). The adapters can turn the plugged mains device on and off, and can at the same time measure the power consumption of the device that is attached. The plugs are called "Circles" and they form a wireless ZigBee mesh network around a coordinator (called "Circle+"). The network communicates with the Controller through a link provided by a USB stick device (called "Stick").

Gateway

The Gateway is a process running in the background, providing two functionalities: (i) Information gathering, reporting power consumption and state of controlled devices; (ii) Device control, used to turn the devices on and off. It is written in Perl using xPL Protocol.[4] In a subcomponent, Application interfaces allow the interoperation of devices (based on possibly different protocols such as ZigBee, X10, Bluetooth, Infrared) and the xPL Protocol. There is the xPL Hub that can bridge various application interfaces and is responsible for passing on the message to the application level for information gathering. It also collects back device control instructions that need to be forwarded to the Environment.

Repository, Collector and Executor

The Repository and the Collector and Executor components are implemented as a Web server that can be accessed with a simple standard protocol, namely, the Jetty[5], HTTP Java-based server, and Representational State Transfer (REST) (Fielding and Taylor 2002) for the communication. Each resource is mapped to a certain resource identifier, usually a Uniform Resource Identifier (URI). For example, assuming the Repository Web server is installed on a local host, the Web service for getting the device's information can be accessed by calling the URI `http://localhost:8080/repository/services/devices`. A client can access these resources and transfer the content using methods that describe the actions to be performed on the resource. The methods are analogous to typical HTTP methods such as GET and POST that describe read and update actions. Each method from the WS interface component calls an appropriate operation from the Data Access Object component, see Figure 8.2. The Data Access Object implements operations that store and retrieve information. It also forms appropriate XML data representa-

[4] `http://xplproject.org.uk/`
[5] `http://eclipse.org/jetty/`

tion needed for other components in the architecture. We use Java Architecture for XML Binding[6] as a technique to map model objects to an XML representation or vice versa. Data Access Object achieves data persistence by using Hibernate framework (Bauer and King 2006) that enables transparent and automatic mapping of the system domain object model into a database. We use MySQL[7] as a relational database management system for all databases.

Scheduler

The Scheduler is a standalone program module written in Scala that is called by the Coordinator whenever there is a need to create a schedule for the following time period. The Scheduler obtains the information about the energy supply and prices from the smart grid via the Coordinator in XML format. Also, it uses the information about the devices and their policies, presented in this format as well. The schedule, created as an XML object, is returned to the Coordinator, and contains a set of actions that should be performed with each device during the next time period.

Coordinator

The Coordinator plays a role of a client to the Repository and to the Controller through the Collector and Executor subcomponent. We use the same technology as for the Repository and Collector and Executor, that is, a Jersey-based client to consume HTTP-based REST Web services requests.

Discussion

The wireless network of plug-in adapters presents a relatively stable topology. We experienced in general a quite good stability during system performance of data collection and command execution. Having a ZigBee network deployed in our building environment, we faced some communication issues due to the radio disturbed environment. In particular, we observed that the microwave, while in working mode, affects the transmission of data through the frequency band of the ZigBee network. In most such cases, the data delivery ratio is lower than 100% (*e.g.,* from 166720 collections, we expected to collect 1000320 measures, but we received 977724 measures), *i.e.,* the information for a particular device or devices is lost. We did not try to solve this issue because the system collects data fairly often so that it does not lose the records of any important state changes. However, one possible solution for

[6] `http://jaxb.java.net/`
[7] `http://www.mysql.com/`

the transmission loss would be to displace the microwave far enough not to interfere with the wireless network of Plugwise devices. Unfortunately, relocating the microwave in our environment was not possible due to space limitations. Another way to improve data transmissions would be to use an acknowledgement process included within the communication (Simek et al. 2011).

Similarly, we noticed another inconvenience when at times the system would not execute the controlling commands for the devices. In fact, there were two reasons for this behaviour. The first relates to the above-described radio disturbances. The other corresponds to the responsiveness of the Plugwise devices themselves. In particular, as the system is collecting data continuously, the execution of a command performed at the same moment as the collection of data was not successful. To resolve the responsiveness issue, we employed programmatically a simple form of reliable messaging with message acknowledgement. In this way, the system re-executes the command until the plug-in adapter is turned into the desired state.

8.4 Evaluation

We have deployed the system in our own offices at Bernoulliborg in order to assess the possible savings obtainable with such a system. The test site consists of three offices occupied by permanent and PhD staff, a coffee corner/social area, and a printer area. The layout is illustrated together with the ZigBee network and the electrical appliances in Figure 8.4. In particular, we include in our testing six available devices (a fridge, a laptop, a printer, a projector, a microwave, and a water boiler). The rated power plate consumptions of the fridge and the laptop are 70 W and 90 W respectively, while that for the printer is 100 W. The projector consumes 252 W when working, while the microwave 1500 W. The water boiler consumes when heating up to 2200 W. Four other sensor nodes are also comprised in the network to strengthen the mesh network connections. We use a set of Plugwise plugs in the same way as described in Section 7.3.

We have used the system over three weeks in the months of October and November 2011, and one week in the month of March 2012, performing measurements from Monday to Friday (as in the weekend there is irregular presence). In particular, in the first 2 weeks (W1-W2) we measured energy use in order to define a baseline. The third week (W3) in 2011 and the fourth week (W4) in 2012, we let our system control the environment in order to measure the actual savings.

We used the REPEAT policy for the fridge (turn on for 15 minutes each hour) and the boiler (turn on for 15 minutes each two hours). The printer used the MULTIPLE policy, and was assigned three jobs over the course of four hours. The microwave used the PATTERN policy, so we used the statistical information from the previ-

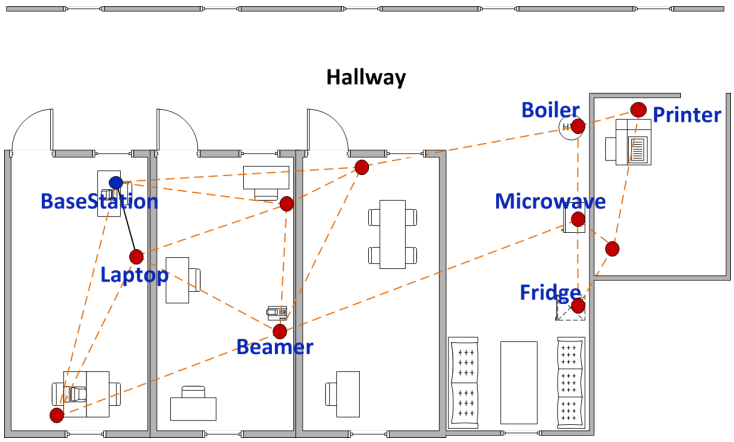


Figure 8.4: Living lab setup.

ously collected data to calculate the expected level of microwave consumption at each hour of the day. The laptop used the TOTAL policy, so it had to be charged for a total one hour during four hours scheduled slots. During week W3, we used the laptop each day. During week W4, we introduced variability of policies usage, so the laptop was used during Tuesday and Thursday. Projector used the STRICT policy to strictly follow the agenda of presentations. During week W3, presentations were given each day from 2 p.m. to 3 p.m. During week W4 presentations were given on Tuesday and Wednesday from 2 p.m. to 4 p.m., thus two hours each.

Next, we present the results in terms of economic savings (due to the varying prices of the smart grid) and of energy savings (due to the introduction of device policies).

8.4.1 Economic savings

The goal of the system is to save money for the office by taking advantage of the smart grid. Therefore, the first evaluation we make is based on taking the energy bill for a week using the system versus a week without it. We have considered two situations for office environments to evaluate the economic benefits of the proposed device scheduling policy: (1) an intelligent office building that interacts with the smart grid Demand-Response tariff service and has small scale renewable installations in its premises that provide power (W3 simulation), and (2) a more ordinary office that has no renewable-based power installation that provide power (W4 simulation) and that benefits only from the tariff differentiation of the smart grid. To obtain a fair comparison in the two simulations, we use the energy prices of the third week (W3) and fourth week (W4) and apply those same retrieved prices for

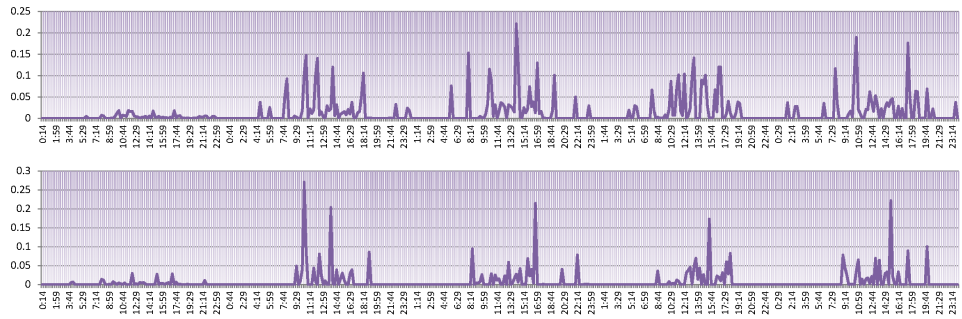


Figure 8.5: Average price ($ per kWh) comparison between non-scheduled (upper chart) and scheduled (lower chart) appliances for each work day (W3 experiment).

the energy consumed in the other two weeks (W1-W2).

In the first set of simulations (office with on site small-scale renewable sources), the situation between each working day of the two weeks (average) without scheduling policies and the week where the policy has been applied is shown in Figure 8.5, where the price of energy ($ per kWh) is shown versus the time of the day (from Monday to Friday). It is interesting to notice the difference in the average price paid for each kWh of energy in the situation without device scheduling and, on the other hand, considering scheduling. The chart is shown in Figure 8.8 (top chart). On average, the price in $ per kWh drops by more than 27% in the two situations. An interesting day where the savings on energy expenses are particularly significant is between the three consecutive Thursdays monitored (October 20th, 27th, and November 3rd). Comparing these three days, the money savings are on average more than 50%. A comparison between the price paid for energy in each hour between the situation in October 27th and November 3rd is shown in Figures 8.6 and 8.7, respectively. In particular, one can see the cut-off of unnecessary energy expenses related to those consumptions that happen during non-working time (late evening or during the night) by devices that are not strictly necessary (most notably the hot water boiler). Another optimisation the system achieves is the most efficient schedule of devices, when the energy generated by photovoltaic panel is more intense and whose cost is generally smaller than energy provisioning on the market.

To validate the scheduling policy, in W4 we consider an office without renewable energy sources (whose price is generally cheaper than energy provision market). Results comparing the day-by-day average price between the scheduling situation and the non-scheduling one are shown in Figure 8.10, while the daily average is shown in Figure 8.8 (bottom chart). One can see that the average price paid when

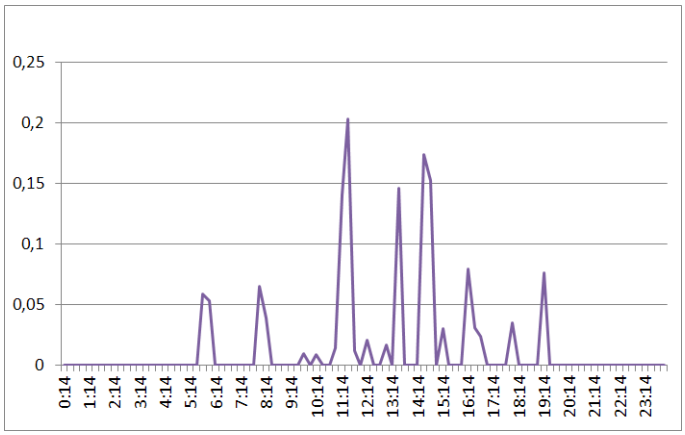


Figure 8.6: Price of energy ($ per kWh) during non-scheduled day October 27th.

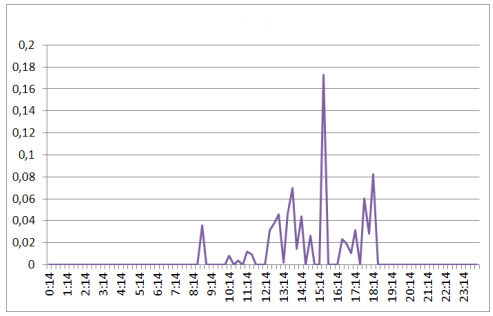


Figure 8.7: Price of energy ($ per kWh) during scheduled day November 3rd.

scheduling is active is usually lower than the non-scheduled situation (*cf.* the continuous and dashed line in Figure 8.8); the overall economic savings between the situation when the schedule is implemented and when it is not is about 22%. The lower savings compared to the W3 experiment are due to the absence of renewable sources in the energy mix of the office, which we have assumed cheaper than the traditional energy market provider prices.

8.4.2 Energy savings

Although energy use reduction is not the primary aim of the system, but rather economic savings based on dynamic pricing, the use of policies for devices alone provides for energy saving in absolute terms. Figure 8.9 (top chart) shows the average energy consumption (kWh) considering the use and the absence of our system comparing W1-W2 and W3 scenarios and Figure 8.9 (bottom chart) compares W1-

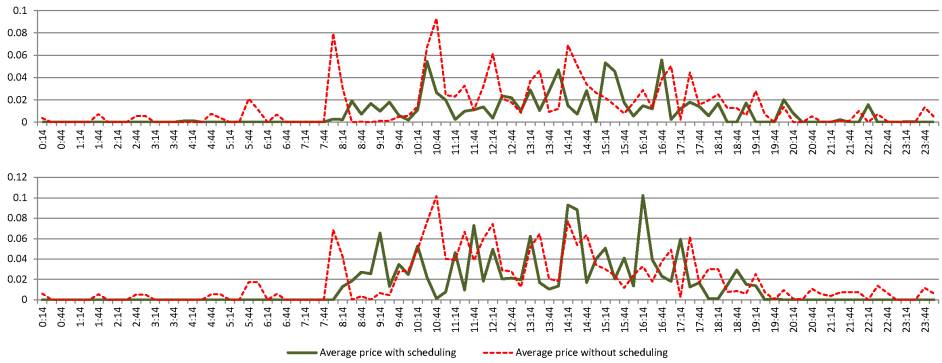


Figure 8.8: Average price ($ per kWh) comparison between scheduled (continuous line) and non-scheduled (dashed line) situations (top W3 experiment, bottom W4 experiment).

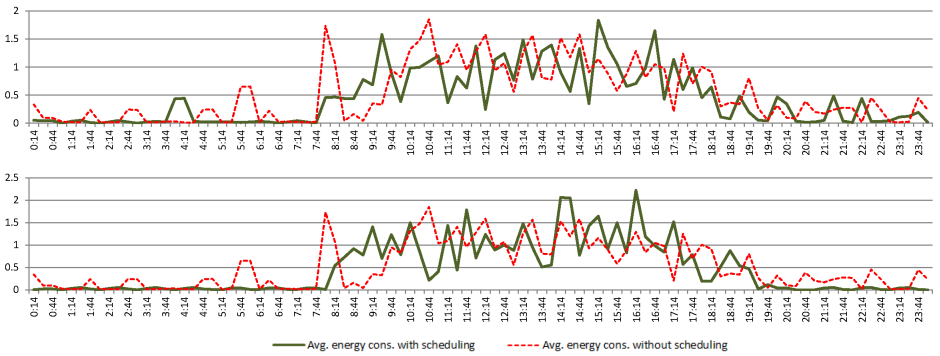


Figure 8.9: Average energy usage (kWh) comparison between scheduled (continuous line) and non-scheduled (dashed line) situations (top W3 experiment, bottom W4 experiment).

W2 and W4 scenarios. The scheduling reduces the consumption of devices that are not used during non-working hours and that do not impact the habits of the users (*e.g.*, keeping the hot water boiler working at night); in addition, the Scheduler tries to use at best the cheap electricity coming from the solar panels during daylight hours. Figure 8.11 visually reinforces the idea of reducing loads when unnecessary among the normal (first upper chart) and the scheduled solutions (the middle and bottom charts): one notices a more compact chart in which energy is used mostly during daytime (8 a.m.-6.30 p.m.) in each day of the week. The average savings of energy consumed between the situation without the scheduling policy and the situation considering it, is more than 15% (W1-2 versus W3 experiment) and about 11% (W1-2 versus W4 experiment), respectively. We ascribe the small difference in percentage to the unpredictable usage of equipment in the living lab between the

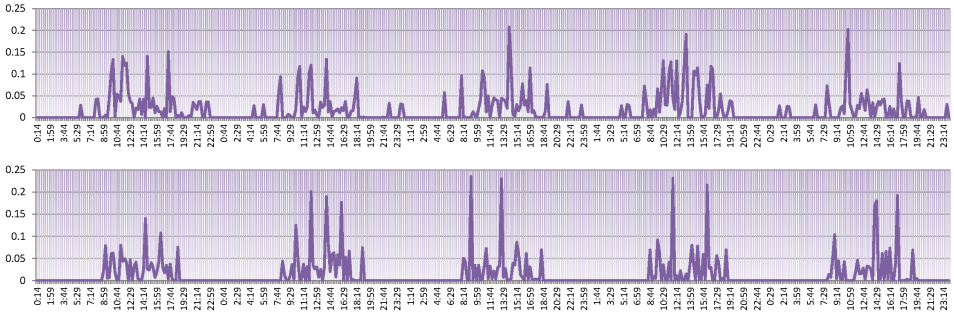


Figure 8.10: Average price ($ per kWh) comparison between non-scheduled (upper chart) and scheduled (lower chart) appliances for each work day (W4 experiment).

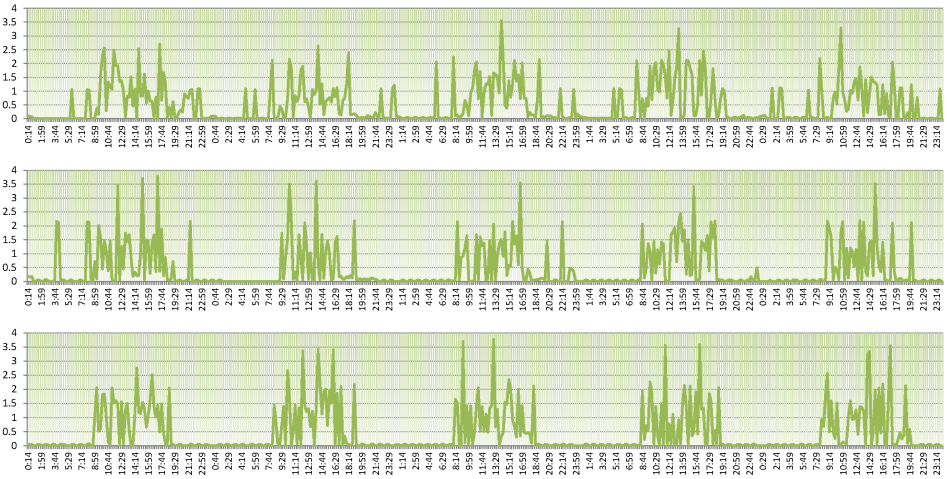


Figure 8.11: Energy (per kWh) comparison between non-scheduled (upper chart) and scheduled (middle and bottom chart respectively W3 and W4) appliances for each work day.

two weeks (*e.g.*, microwave use).

8.4.3 Remarks

We remark that the system itself consumes energy to operate; it consists of 10 Plugwise devices and one desktop computer that respectively consume a maximum power of 1.1 W and 365 W, respectively. The value of the plugs is insignificant with respect to the overall consumption. As for the computer, a few remarks are in order: firstly, the optimisation program does not need to run on a dedicated computer, so it could add little consumption to the already active computers. Secondly, in a real

operational environment, the system would schedule many more devices; thus, its energy consumption would be amortized over larger savings. For these reasons, we have not included these energy consumptions in the current evaluation.

Chapter 9

Composing applications ready for deployment

Consider the two systems we built for coordination in ubiquitous computing environments. Both represent distributed applications that consist of several components. The components offer their capabilities as services, resulting in applications composed of services. The applications are usually composed manually, as in our cases, or with some predefined scripts. In reality, either way makes the composition process a difficult problem due to several factors. First, although each service is responsible for a separate aspect of the systems, the services are highly interrelated. Second, each service may have multiple versions each of which includes a different set of requirements for communication, exchange of information, and capabilities of other services. Third, each service may have multiple instances running in the same setting – a real situation rather than a vision. To illustrate this better, just imagine the system we build for coordinating offices. There are 300 offices in the building, distributed on four floors. A single instance of a service cannot deal with such scaling of the environment. This implies that the number of services in the configuration of the system for an actual deployment may vary and increase, which is a fourth factor.

A solution to this problem is to build distributed applications automatically, that is, automating the process of service composition. Luckily, we can resort to planning to accomplish this. The main assumption that enables planning to be used for composing services is that they are discoverable and need to be associated with semantic annotations that define their functionality. There are two possibilities for this, and we look at each in the following.

Nowadays, services usually reside on the Web or in the Cloud. Web services are distributed on the Internet, publicly available through standard protocols (*e.g.,* HTTP), and registered in some repository, such as the Universal Description, Discovery and Integration (UDDI) registry. The main issue with Web services lies in the lack of a consistent semantic annotation, such that it is feasible in practice. Even though there are existing ways to describe Web services (*e.g.,* SOAP, WSDL, OWL-S), the reality of Web services is that they are associated only with syntactic

specifications and free-text descriptions, leading to the consideration of public services as nothing more than data sources (Fan and Kambhampati 2005).

Cloud services are not necessarily accessible over a network that is open for public use. In most scenarios, Cloud services are only accessible by corporations providing them with greater control and privacy. Services in such well-controlled environments have different characteristics from the services rendered for the Web. This includes services to be more structured and to have annotations given by the providers using a consistent ontology. There are also cases where machine-interpretable annotations may be provided. In fact, corporations tend to make use of well-established standards and best practices they gain in the domain of service-oriented architectures to support a standardised way of access to Cloud services. All this foregrounds the possibility to make service composition feasible.

We expect that distributed applications ready for deployment, such as those for ubiquitous computing environments, will evolve as Cloud applications rather than Web applications. While various planning techniques are already well studied for building the latter ones (*i.e.,* Web service composition), the use of planning for composing services for the Cloud is scarce. Though there are similarities between the two, for example, the satisfaction of interdependencies between services, the composition of Cloud services may in addition involve configuration processes that enable correct service instantiations, valid state transitions of services, *etc.*

We introduce an approach based on HTN planning that automatically composes applications ready for Cloud deployment. We suggest that HTN planning is suitable for this due to its rich domain knowledge, modularity, recursive structures, and natural representation of causality. We use an existing formal model to describe a deployment problem, and then we propose a strategy to create an HTN planning problem from the deployment one. Further, we implement our approach using the **SH** planning system and perform a set of experiments to assess the feasibility of our approach.

In addition, we look at Web service composition too. We analyse in more details the challenges of Web service composition, and derive a general model for composing Web services via planning. Based on this general model, we look at a concrete relationship between Web service composition and HTN planning. We then discuss the state of the art, aiming to identify the shortcomings of current HTN-based approaches. In what follows, we use 'service' and 'component' interchangeably.

9.1 Composition of Cloud applications

Cloud computing brings new interesting perspectives to the conventional way of creating, manipulating, and using everyday applications (Hayes 2008, Vaquero

et al. 2008). The applications are no longer installed and run on a single machine, but they are composed of assorted services that are deployed and distributed on different machines of Cloud infrastructures. A class of problems is associated exactly with the process of composing and deploying such modern applications. Given some initial configuration of a Cloud infrastructure in terms of already deployed services, a set of deployment actions, such as start and stop a service, and a desired application, a deployment problem consists of finding a sequence of deployment actions over services that compose the desired application.

While the deploying aspect of this process is already fully automated, the composition of application components is yet to be improved, being still performed either manually or semi automatically with some predefined scripts. The scripts may ease the process to a certain degree, but their use is limited as they are exclusively dedicated to specific services and applications. Moreover, even a small number of components can make the composition process already strenuous and difficult. Thus, the difficulty increases further with the number of services to be configured, especially when the services are delivered in different builds and releases with different compatibilities among each other. The process of satisfying interdependencies between services then can no longer be performed manually or with the mainstream tools.

We use HTN planning to address the problem of automated composition of application components. We propose a strategy to create an HTN planning problem from a deployment problem. We use a so-called Aeolus model (Cosmo et al. 2012) to define the deployment problem. In this model, components are resources of various kinds that require and provide functionalities through ports. Requiring and providing functionalities implies establishment of interdependencies between components. A requested application is realised by a sequence of low-level actions, such as create instance, start instance, bind port, and so forth. We demonstrate the applicability and feasibility of this approach through an experimental evaluation, and we show that general-purpose planners, such as an HTN one, can be likened to specialised ones to a certain degree, contrary to the results presented in (Lascu et al. 2013).

9.1.1 Deployment model

One way to define the problem of configuring and deploying applications on the Cloud is by using the Aelous model (Cosmo et al. 2012). The main element of the model is a *component*, describing a manageable resource that provides and requires functionalities. Through the use of state machines, the Aeolus model provides a way to encode specific components declaratively by specifying how functionalit-

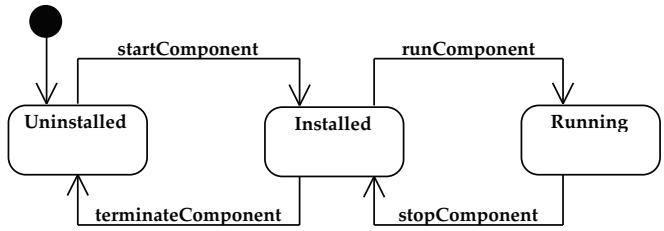


Figure 9.1: FSM depicting the state transitions of a component specified in UML.

ies are accomplished. We consider a component as the Finite State Machine (FSM) shown in Figure 9.1. The FSM defines the *state transition* processes of a component, *i.e.,* the states and the order in which a component can transition from one state to another. A component is initially in an *unintalled* state. Upon start, it transitions into an *installed* state, and then to a *running* state. State transitions are accomplished using *deployment actions*. For example, given some component in its initial state, it is installed by invoking the *startComponent* action.

In most cases, however, a component can transition in some state only if the functionalities that particular state requires through *require ports* are communicated by components that can provide them through *provide ports*. We can observe such transitions in configuration patterns (see Figure 9.2). A pattern contains a set of components interrelated among each other through the ports on the level of states. The components are abstract, meaning that they will be replaced by concrete components, or *instances*, at runtime. A single configuration pattern therefore defines a number of actual compositions.

A *component* c is a 5-tuple $\langle Q, q_0, U, P, R\rangle$, where Q is a finite set of states, q_0 is the initial state, $U \subseteq Q \times Q$ is the set of state transitions, P is the set of provide ports, and R is the set of require ports. We denote the set of all available components as C, and the set of all ports as F. The set A consists of the deployment actions used upon the elements in C and F. A *configuration* D is a tuple $\langle C, I, \phi, B\rangle$, where C is a set of available components, I is a set of currently deployed component instances, ϕ is a function that associates $i \in I$ with a pair $\langle c, q\rangle$, where $c \in C$ and $q \in Q$ is the current component state; and $B \subseteq F \times I \times I$ is a set of bindings.

A deployment problem consists of an initial configuration, a set of deployment actions, and a request for a new configuration (*i.e.,* application). The solution to the problem is a deployment run representing a sequence of deployment actions on components that, when deployed, produce the required configuration.

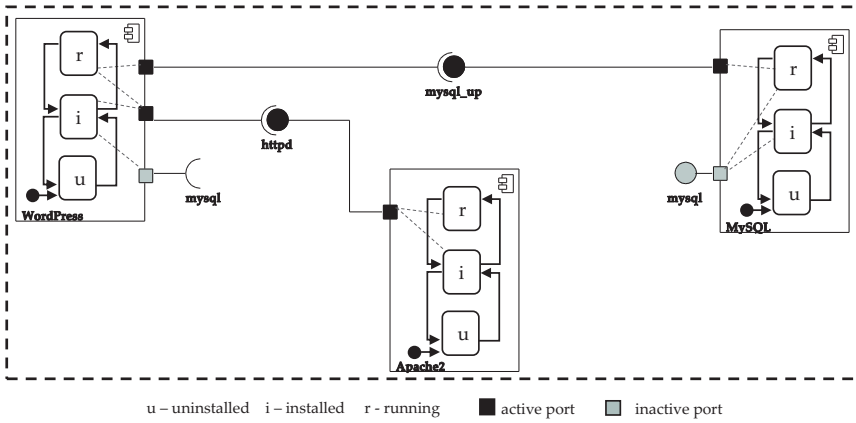


Figure 9.2: Example of a pattern for a *WordPress* application.

9.1.2 Hierarchical planning domain model

We introduce an approach to create an HTN planning problem from a deployment problem. We provide a running example that helps in demonstrating the structures we encode in the domain model. We use the Hierarchical Planning Definition Language (HPDL) (Fernández-Olivares et al. 2006) when describing the planning structures. In the following, we refer to a state transition that does not depend on any functionality provided by other components as *simple transition*. Otherwise, we use the term *complex transition*.

A running example

Figure 9.2 graphically represents an Aeolus pattern for composing a WordPress[1] application in a running state. The main and top-level component represents *WordPress*, which is a popular blogging system. *WordPress* operates using several software services among which essential ones are a Web server and an SQL database. The application requires a database to store all blog information (*e.g.,* posts, comments, meta-data). The most commonly used database is MySQL, but other databases, such as MariaDB and Percona Server, are compatible too. A recommended server is Apache, but any other server that supports PHP and MySQL is suitable too. We use *MySQL* and *Apache2* as components that *WordPress* depends on.

[1] `https://wordpress.org/`

Components, states and ports of components

We encode components, instances, ports as domain types `component instance port`, which are all subtypes of the type `object`. In fact, each component type, such as *WordPress* is represented as an object of type `component`.

While FSMs associate components with states abstractly, component instances are the ones to be in a specific state at planning time. We encode an instance state using a predicate "*(state instance)*", where *state* is a string representing the type of an FSM state, and *instance* is a variable representing the component instance. An example of a *WordPress* instance *w1* in an installed state is `(installed w1)`.

A component state may be associated with require and provide ports. To represent the association of a port to a state, we use a predicate "*(statePort component port)*", where *statePort* is a string representing the type of port in a specific state, *component* is a variable representing the type of component that requires or provides a port represented by the variable *port*. For example, if *WordPress* requires the *httpd* port in the installed state, we encode it as `(installed-require wordpress httpd)`. Such knowledge holds for all instances of the respective component. These predicates are therefore grounded in the initial state and static during planning.

Creating new component instances

One of the features of the composition of Aeolus applications is that one or more component instances must be created from existing (abstract) components. Contrary to the approach taken in (Sohrabi et al. 2013), where an assignment expression in the precondition of an operator is used to create a new object, we address the creation of new uninitialised instances using a *domain function*. This function returns a number that we use to represent instance variables in a special predicate `(instance ?iNum - number)`. The `instance-number` function practically serves us as a counter to keep track of the current value that can be assigned for new instances. The domain function does not take arguments. We use an additional predicate `(type ?iNum - number ?c - component)` to associate the instance with a particular component. We increase the instance number, and assert the association by manipulating the effect of the operator that creates new instances as showed in the following encoding.

```
(:action createInstance
 :parameters (?c - component)
 :precondition ()
 :effect (and (instance (instance-number))
              (type (instance-number) ?c)
```

```
            (increase (instance-number) 1)))
```

Deployment actions

In addition to `createInstance`, we consider the actions that accomplish simple transitions. These are the deployment actions, including the binding ones. The binding actions are responsible for low-level binding of ports – the require ports are bound to the provide ports. We encode all these actions as HTN operators. The parameters of operators corresponds either to a component instance variable or to variables of a port and two instances (in the case of binding actions). The preconditions and effects of each operator capture the semantics of the respective action. The following is an operator that corresponds to the *startComponent* deployment action, which makes the state of a instance to become installed and activates all the ports associated with the installed state of the component which the current instance belongs to.

```
(:action start
 :parameters (?i - instance)
 :precondition (and (not (installed ?i)))
 :effect (and (installed ?i)
              (forall (?p - port) (when
                      (and (installed-provide ?c ?p)
                           (type ?i ?c))
                      (active ?p ?i)))))
```

Other deployment actions are encoded similarly. As for the binding ones, the `bind` operator creates a binding between the provide port of some instance and the require port of another one, and the `unbind` operator deletes an already established binding between two components' instances.

Configuration processes

Although each different type of an application has its own installation and running configuration pattern, the process of configuring applications is general and can be abstracted away. Let us detail how we can accomplish that.

The process of configuring an application requires satisfaction of the dependencies to functionalities provided by components. Let us assume that an instance in an uninstalled state cannot have requirements to be satisfied. We may then consider two abstractions for complex transitions of components. The first abstraction refers to acquiring a component functionality in the installed state, while the second one refers to establishing a functionality in the running state. We point out that complex

transitions representing other configuration types can be easily incorporated in the current domain model with minor modifications (see Section 1.1.2).

We encode each abstraction as a task in the domain model, namely `install` and `run` tasks. Each method of these tasks encodes a specific case. One such method involves port activation. If a component state is associated with one or more require ports, the **port activation** process makes sure that the need of the current instance for specific functionalities is addressed. That is, if the current component instance has require ports that are not active, the method first activates each port and calls recursively its corresponding task until all necessary ports are activated. The actual process of port activation is encoded in a separate task. The task not only activates a required functionality, but also finds and installs (or runs) a component instance that provides that functionality. An instance with active require ports can then use the functionalities of other components with active provide ports. This is accomplished by another method that involves port binding. The process of **port binding** binds require ports to appropriate provide ports. For this process, the method depends directly on the binding actions. Once we have methods that involve port activation and binding, we can proceed to the method that deals with the case when all require ports are active and bound. To address the satisfaction of all require ports, we use a *forall* expression in the method for both tasks, `install` and `run`. The following expression is used for the `install` task.

```
(forall (?p - port) (and (installed-require ?c ?p)
                         (bound ?p ?i ?i1))))
```

After this constraint check, we are ready to start or run an instance. In the case of the `run` task, when running an instance, we have to deactivate the ports that will be no longer provided by the instance in the installed state. The process of **port deactivation** is accomplished using a separate task with multiple methods. Each method represents a different case to be handled, such as a provide port that is bound but needed for the running state, a provide port free to be unbound, *etc*. The port deactivation task uses port unbinding. The process of **port unbinding** is more complex than the binding one, and requires checking for constraint violation. That is, we have to take care of active provide ports bound to active require ports. We use a separate task for this process, that is, `unbindPorts`. This task does nothing when the port is bound and needed for the next transition. When all necessary constraints are satisfied, it unbinds a specific port and recursively calls itself, shown in the following encoding. Being a recursive task, it includes a base case that performs phantomisation (Georgievski and Aiello 2015a).

```
:tasks (sequence (unbind ?p ?i ?i1) (unbindPorts ?i))
```

There are methods in the `install` and `run` tasks that deal with the case when there are no required functionalities for an instance. This means that we have a simple transition which can be handled by installing the component instance directly. In the case of running an instance, we invoke the port deactivation task to ensure a valid transition to the running state.

The modelling of the transitions from a running state to an installed state and further to an uninstalled state is analogous to the encoding of the tasks we described so far.

One of the features of these kinds of compositions is that a cycle may occur between states of different component instances. That is, an instance is expected to provide a functionality at a specific point in the composition, but it is not possible because at the same point the instance is required to change its state (Lascu et al. 2013). We address this feature using the process of **instance duplication**. Instance duplication deals with such cycles by creating as many instances of the same component as needed, and deploying them in different states at the same time. We encode instance duplication as a separate method. The method makes sure that the current component instance is in a specific state and it has at least one provide port bound. Consequently, a new component instance is created either in an installed state or in a running state, depending on the type of configuration.

Algorithm 6 shows the high-level steps of the strategy we described for the creation of an HTN domain model.

Algorithm 6 Transformation of an Aeolus model into an HTN planning domain model

Input: a set of components C, a set of deployment actions A
Output: HTN planning domain model $\langle O, T \rangle$

1: Encode component, instance, port as types
2: Choose $c = \langle Q, q_0, U, P, R \rangle$ from C
3: **for** $j = 1$ to $|Q|$ **do**
4: Create state predicate and port predicates for q_j, $q_j \in Q$
5: **end for**
6: Encode an operator o for creating instances
7: **for** $j = 1$ to $|A|$ **do**
8: Encode a_j as an operator o_j, $a_j \in A$
9: **end for**
10: Ask the user questions regarding the configuration processes in $\langle C, A \rangle$, and encode the corresponding tasks

9.1.3 Deployment-based HTN planning problem

A deployment problem P^D is a tuple $\langle D_0, A, G\rangle$, where D_0 is the initial configuration, A is the set of deployment actions, and G is the requested configuration. δ is a satisfying deployment run for P^D if and only if δ is a sequence of deployment actions that transform D_0 into G. A requested configuration, G, is achievable if and only if there exists at least one satisfying deployment run for it.

Given a deployment problem P^D, we define the corresponding deployment-based HTN planning problem $\mathcal{P}$ according to Definition 5.2, where 1) s_0 is the initial state consisting of a list of the following ingredients derived from D_0: components and ports as objects, component states, currently deployed instances, the current state of deployed instances and bindings as the special predicates we defined in the HTN planning domain model. s_0 also contains a domain function initialised to 0. 2) tn_0 is the initial task network encoding the requested configuration G; 3) O is the set of operators that represent actions in A, and T is the set of tasks derived from the configuration processes with respect to Algorithm 6. A plan π is a solution to $\mathcal{P}$ according to Definition 5.3.

9.1 Theorem. *Let P^D be a deployment problem and $\mathcal{P}$ be the corresponding HTN planning problem. If a requested configuration G is achievable, then there exist a plan π for $\mathcal{P}$.*

Let δ be a satisfying deployment run for P^D such that G is achievable. Under the assumption that the user provides reasonable answers – there is a correspondence between P^D and $\mathcal{P}$ as defined previously, then there must exist a solution for $\mathcal{P}$.

We can now obtain that the solution of the deployment-based HTN planning problem is a deployment run for the corresponding deployment problem.

9.2 Theorem. *Let P^D be a deployment problem and $\mathcal{P}$ be the corresponding HTN planning problem such that Theorem 9.1 holds. We can then construct a sequence of deployment actions based on π that is a satisfying deployment run for P^D.*

Let us present a constructive proof for which we consider the deployment problem P^D shown in Figure 9.2. Let $\mathcal{P}$ be the corresponding deployment-based HTN planning problem. Furthermore, consider the following plan for $\mathcal{P}$: [createInstance(w0), createInstance(a1), start(a1), bind(httpd,w0,a1), start(w0), createInstance(m2), start(m2), run(m2), bind(mysql-up,w0,a2), run(w0)]. We can construct a deployment run in which the actions from the plan are deployment actions. The resulting deployment run is a satisfying deployment run for P^D.

9.1.4 Evaluation

We have two main objectives with our experimentation: 1) to evaluate the applicability of our approach for composing Cloud applications, and 2) to counter the

prior negative results of the performance of general-purpose planners in the composition of Cloud applications (Lascu et al. 2013).

To address our objectives, we generate deployment problems of increasing number of components varying from 3 to 220 components, resulting in more than 50 problems. We apply our approach to create the corresponding HTN planning problems, and examine the performance of **SH** on them. The HTN planning problems are constructed from deployment problems of varying difficulty. For example, the difficulty of a problem can be increased if there is a need for instance duplication. Also, for **SH**, an HTN problem can be more difficult if the requested configuration appears deeply in the search space. To that end, we construct two cases of deployment problems mainly following the test pattern provided in (Lascu et al. 2013). For both test cases, we use a set of components $c_1, \ldots, c_n$, where each c_i has require and provide ports as follows. Given that we want to have the rightmost component c_n in its running state, the dependencies between components will require to first create instances for components from c_1 to c_n, then to perform transition from uninstalled to installed state in the reverse order of component instances, and finally, to transition from installed to running state in the order from c_1 to c_n. We modify the second test case in such a way to require instance duplication. In particular, we randomly select several components and, for a selected component c_i, we remove the activation of a provide port p_i^1 from its running state. The removal requires another instance of c_i to be created so as to satisfy the requirements of c_{i-1} and c_{i+1}.

We show a subset of our results in Table 9.1. The left-hand side of Table 9.1 shows the results of the first test case without instance duplication, while the right-hand side shows the results with instance duplication. Columns three and six show the time in seconds needed to find a solution. For each problem, we show the plan length as an indication of the difference between the number of operators creating instances and the number of other deployment actions. In the case without duplication, the number of generated instances equates to the number of components, while in the latter case it is strictly greater than the number of components. With the creation of a new instance, we increase the size of the state by adding two predicates, and modify the state by updating the domain function.

All problems are solved within 17 seconds. When the number of components is larger than 120, the need for instance duplication degrades the performance of **SH** as compared to the case without instance duplication. However, in typical applications, there are hardly any scenarios with more than 100 components for which case **SH** can find a solution in about 2 seconds with and without instance duplication. The results also show that the planner runs out of memory when the problem has more than 200 components with instance duplication and 220 components without duplication. This is mainly due to the implementation of the core part of

Table 9.1: Evaluating the applicability of our approach by using **SH** under increasing problem difficulty ("OM" signifies "out of memory").

Without duplication			With duplication		
Problem	**Plan length**	**Time (sec)**	**Problem**	**Plan length**	**Time (sec)**
3	12	0.077	3	16	0.017
6	27	0.032	6	35	0.004
10	47	0.041	10	55	0.012
20	97	0.193	20	109	0.046
30	147	0.226	30	171	0.113
50	247	0.354	50	287	0.389
70	347	0.784	70	399	0.898
100	497	1.957	100	577	2.34
120	597	3.112	120	693	3.871
150	747	5.791	150	863	7.182
180	897	9.625	180	1037	11.916
200	997	12.897	200	-	OM
220	1097	16.918	220	-	OM

SH, which employs recursion (in these test cases, the number of recursive calls increases rapidly), and the need for creation and maintenance of a large set of objects.

These results also address our second objective and show that general-purpose planners can exhibit a satisfactory performance in composing Cloud applications. Compared with the results of the two general-purpose planners reported in (Lascu et al. 2013), our HTN planner outperforms both planners significantly. Compared with the specialised planner, our planner falls behind the specialised one only after a reasonably high number of components.

9.1.5 Related work

The communities of AI planning and cloud computing have explored the automated composition of components.

Planning. Many studies use automated planning to compose Web services (*e.g.,* (Kaldeli et al. 2011)), and to automatically generate information flows (Riabov and Liu 2005, Sohrabi et al. 2013), which is an analogous problem to Web service composition. Among those studies, HTN planning is employed to represent and compose Web services in multiple approaches, which we discuss in Section 9.2. The most common one translates the service knowledge from Web Ontology Language for Services (OWL-S) (Martin et al. 2007) to HTNs (Sirin et al. 2004). The

main difference between OWL-S and Aeolus lies in that the latter is envisioned for capturing deployment processes of distributed Cloud applications, while OWL-S is specifically designed to support the discovery, composition and monitoring of Web services.

There are also attempts to use automated planning for composing Cloud applications. Arshad et al. (2003) describe a deployment problem of software components, and use general-purpose temporal-based planner to find the most optimal plan with respect to plan duration. Lascu et al. (2013) represent a deployment problem using a simplified Aeolus model, and develop a specialised planner to search for a solution. While the former study does not define the planning problem on any formal ground, we use the simplified Aeolus formal model as in the latter study to derive our HTN planning problem. Contrary to (Lascu et al. 2013), where domain-related processes and features are implemented and embodied in the planning process, we use a general-purpose HTN planner, and encode the specific knowledge into the domain model.

Cloud computing. Juve and Deelman (2011) propose a system that provisions, configures, and manages deployments of virtual machines in a Cloud. They also describe their experiences using the system to provision resources for scientific workflow applications. Kirschnick et al. (2012) describe an architecture that enables automatic provisioning of services in the Cloud, the language used to describe the services to be deployed, and how a new service is managed.

While there is also a number of automated tools for Cloud creation and Cloud management available, we here list the most popular ones. Chef[2] automates build, deploy, and manage processes on a Cloud infrastructure. CFEngine[3] is a configuration management system that provides a framework for automated management of an IT infrastructure. Puppet[4] makes fast and repeatable changes, and automatically enforces the consistency of systems and devices across physical and virtual machines, both on the premise or in a Cloud. AWS CloudFormation[5] gives developers and systems administrators an easy way to create and manage a collection of related AWS resources, provisioning and updating them in an orderly and predictable fashion.

There is also open-source software for creating public and private clouds. Eucalyptus[6] is software for building private clouds that are compatible with AWS APIs.

[2] `https://www.chef.io/chef/`
[3] `http://cfengine.com/`
[4] `http://puppetlabs.com/`
[5] `http://aws.amazon.com/cloudformation/`
[6] `https://www.eucalyptus.com`

Apache CloudStack[7] is software designed to deploy and manage large networks of virtual machines as a highly available and highly scalable cloud computing platform. OpenStack[8] is software that controls large pools of compute, storage, and networking resources throughout a data centre, managed by a dashboard or via the OpenStack API.

9.2 Web service composition

Web services are software components that implement specific business logic, and are distributed over the Web to be used as Web resources for machine-to-machine interaction. For instance, travel agencies may provide a number of Web services, such as booking a flight ticket, reserving a hotel, renting a car, or organising sightseeing. The interaction is usually initiated by a client request which has to be satisfied by the functionalities that Web services offer. However, in cases when no single service can accomplish the request, a composition of several Web services might give a value-added functionality, and provide a way to request satisfaction. For example, a service to arrange a complete trip to some tourist destination might be of an exceptional use to the commercial travel agencies, and thus, it will not be offered as a Web service.

The AI community tries to automate the process of Web service composition by viewing the composition problem as a planning problem (Aiello et al. 2002, Sirin et al. 2004, Lazovik et al. 2004, Dustdar and Schreiner 2005, Kuter et al. 2005, Medjahed and Bouguettaya 2005, Klusch and Gerber 2005, Sohrabi et al. 2006, Paik and Maruyama 2007, Kaldeli et al. 2009, 2011). The general assumption is that planning operators correspond to functionalities of Web services, while the goal, in the simplest example, is aggregated from the client request. If the client's objective, for example, is not only to reserve a hotel, but to arrange a complete trip, which includes also booking a flight, renting a car, and sightseeing, then, definitely, the complexity of services and their composition becomes an interesting and challenging task.

The environment of Web services offers more exciting challenges that make the effective selection and composition of services far from being plain and straightforward planning processes. In particular, Web services exist in a *dynamic* environment in which the availability of services is not guaranteed. This behaviour reflects the availability of information which, on the other hand, is assumed by planners to be complete and obtainable before the planning process is initiated. Furthermore, the environment of Web services favours techniques that are able to deal with *uncertainty* in terms of 1) incomplete information about the initial state; 2) uncertainty

[7] `http://cloudstack.apache.org/`
[8] `http://www.openstack.org/`

over the many possibilities for completion of missing information by invoking some sensing services at planning and/or execution time; 3) non-determinism caused by failed invocations of Web services (*e.g.,* renting a car is not viable at the moment of invocation), a service not responding at all, a service yielding an undesired outcome (*e.g.,* booking a flight provides only business-class tickets); 4) services that show unexpected behaviour (*e.g.,* Byzantine failure). Moreover, *complex goals* possibly in the form of a workflow or conditioned with some organisational regulations or augmented with user preferences are the norm rather than exception. Finally, the *high cardinality* of the set of Web services available on the Web implies a large space to be searched by a planner.

9.2.1 WSC via planning

A general approach towards composing Web services via planning without restrictions to specific external and internal service representations, or a particular planning technique has the following steps. A Web service is usually described in some (external) language. The WSC problem consisting of such services is provided to a translator that creates an appropriate (internal) representation, that is, a planning problem. Consequently, the planning problem is given to a planner to search for a solution. If there is a solution found (*i.e.,* a plan), it is passed for execution and monitoring for potential faults. In case of a fault, appropriate actions are taken.

- **Service description**: The description of Web services offered to the global market usually consists of three parts. The first part refers to the information about the data transformation during the execution of a service. The information is presented in form of input, output and possibly exceptions. The input contains the information required for service execution, while the output presents the information the service provides after its execution. The second part refers to when and how a service transforms the world. This part consists of preconditions, that is, requirements that must be satisfied for the service to be invoked, and postconditions, that is, physical changes to be made to the world. The last part contains the non-functional properties of a service, such as cost, reliability, and service quality.

- **Translator**: Services described in a standard Web service language appear to be hard to handle by planning systems unless they are translated into an understandable form. The translator accepts service descriptions and converts them into formal and unambiguous encoding. The result of the translation is a planning problem. In fact, this component enables the relationship between Web service composition and automated planning.

- **Planning system**: It takes the planning problem and tries to find a solution. Many planning systems distinguish between world-altering and sensing actions. The former can change the world when executed, while the latter cannot modify the state, but only acquire additional information needed to support the planning process. The most common approach is to perform off-line planning, that is, to simulate the execution of world-altering actions, and to do sensing. The solution, if it exists, consists of world-altering actions only. Many planning systems make several assumptions while planning. These assumptions simplify the planning process, but impose restrictions about what might happen in the world and distance further from the reality. The assumptions are:

  A1: The world is static – it can be modified only by the actions resulted from the planning process, and not by some external agent or event. All information about the world is expected to be valid till the end of the execution.

  A2: Sensing actions succeed – the execution of a sensing action will always return the acquired information.

  A3: Sensing actions are repeatable – the first sensed information is assumed to be valid for each action (service) further in the planning process.

  A4: No changes are made to services – service's functional properties are constant during the planning and execution processes.

- **Execution Monitoring and Contingency Handling**: Considering that the world is dynamic and uncertain, the execution of actions might not proceed as expected. A contingency may be inconsistent sensed information, failures of service invocations, timeouts, or unexpected change in the world. These observations suggest that the problem of Web service composition should not be tackled decoupled from the process of action execution. Monitoring of execution and contingency handling appears to be suitable to address the aforementioned issues. Execution monitoring checks the validity of off-line calculated actions when executed and, in case of contingency, reacts appropriately. For example, if the execution time of some service takes too long, then it might be possible to proceed with the execution of subsequent actions. Other types of contingency may require repair of the existing plan, or even planning from scratch.

9.2.2 WSC problem as an HTN planning problem

We now make a concrete and strong connection between WSC and HTN planning by choosing OWL-S as a service description language upon which we define the problem of Web service composition and its corresponding HTN planning problem. OWL-S (Martin et al. 2007) is a Web ontology (Horrocks et al. 2003) for Web services used to support automated discovery, enactment and composition of Web services. The OWL-S ontology has three components: service profile, process model and service grounding. The service profile indicates the purpose of a service, and comprises the elements of part one and part three described in 'Service description' step in the framework. The process model indicates how to accomplish the service purpose, how to invoke the service, and what happens after the service execution. The service grounding specifies the way of interaction with the service, including a communication protocol.

The similarity of OWL-S with HTN planning lies in the services that OWL-S perceives as processes. OWL-S differentiates three classes of processes: atomic, simple and composite. An atomic process has no sub-processes, has a grounding associated with it, and can be executed in a single step. Then, a simple process provides an abstraction for an existing service, and has no associated grounding. Finally, a composite process consists of other processes via the control constructs.

The services described in OWL-S need to be encoded in corresponding HTN elements. Intuitively, each atomic process is translated to an operator, and each simple and composite process is translated to a method (Sirin et al. 2004). If we consider that $\mathcal{P}^{\mathcal{W}} = (s_0, K, C)$ is a WSC problem described in OWL-S, where s_0 is an initial state of the world, K is a collection of OWL-S process models, and C is a composite OWL-S process defined in K, then the following relationship could be established (adopted from (Sohrabi 2013)).

9.3 Definition (WSC relationship to HTN planning). *Let $\mathcal{P}^{\mathcal{W}} = (s_0, K, C)$ be an OWL-S WSC problem. Then, the sequence $p_1, \ldots, p_n$, where each p_i is an atomic process defined in K is a solution to $\mathcal{P}^{\mathcal{W}}$ if and only if $t_1, \ldots, t_n$ is a solution to an HTN planning problem $\mathcal{P} = (Q, O, M, tn_0, s_0)$, where*

- *Q, O, M are generated by an OWL-S to HTN translation for the OWL-S process models K,*
- *tn_0 is generated by an OWL-S to HTN translation for the OWL-S process C, and*
- *each t_i is a primitive task that corresponds to an atomic process p_i defined by some OWL-S to HTN translation.*

9.2.3 Overview of planners

With the general model of planning for Web service composition, and the concrete relation between the WSC OWL-S problem and HTN planning problem, we can classify and examine the studies employing HTN planning for WSC. This enables us to identify the shortcomings of current HTN-based approaches to WSC.

Table 9.2 summarises the studies with respect to indicators extracted from the model (a detailed discussion on each study can be found in (Georgievski and Aiello 2014)). Some indicators are associated with ratings. The ratings range from '★', indicating limited focus or limited support for a respective indicator, to '★★★', specifying comprehensive focus or extended support for the corresponding indicator. If a cell contains '✕', it means that the planner does not support the indicator under inspection. If a cell is empty, it denotes that we were not able to extract the information for the respective indicator from the literature. **Service description** provides the language for describing Web services assumed by the study being analysed, while **Translation** gives the dual information. First, it indicates how well and exactly the translation process is described, and second, which format is the Web service description translated to. **HTN model** tells whether state-based HTN or plan-based HTN planning is employed, and which HTN planner is used for the implementation of the taken approach. Beside the extent to which it is supported, **Sensing** may indicate whether the execution of a sensing action blocks the planning process, and whether sensing actions are performed during planning or they may be interleaved with world-altering ones during execution. **Assumptions** concern the degree of assumptions made to guarantee a successful composition with respect to composing, sensing and executing actions. **Contingencies** refers to unexpected behaviour of a composition at execution time, including Web service failures or time outs, and events or information changes made by some external agents. Each approach is evaluated with respect to the extent to which the support is implemented, and the type of contingency the approach can handle.

In Table 9.2, we group the studies into two categories. In the upper part, we analyse approaches that employ HTN planning exclusively in the attempt to solve the problem of Web service composition. In the lower part, we observe approaches that combine HTN planning with another technique, such as description logic and constraint satisfaction, to compose services.

Most of the approaches assume OWL-S description of Web services, and provide sound translation algorithms to an appropriate internal representation. With respect to the HTN model, all approaches but one employ state-based HTN planning. From the state-based HTN approaches, one uses the SIADEX planner, one the **SH** planner, while the rest exploit SHOP, its Java version (JSHOP), or its successor

Table 9.2: Summary of HTN-based approaches to Web service composition.

Study	Service description	Translation (Representation)	HTN model (Planner)	Sensing (Properties)	Assumptions	Contingencies (Types)
(Wu et al. 2003, Sirin et al. 2004, Kuter et al. 2005)	DAML-S OWL-S	★★★ (SHOP2)	state-based (extended SHOP2)	★★ (blocking/non-blocking, during planning)	A1-A4	× (not discussed)
(Madhusudan and Uttamsingh 2006)		★ (SHOP)	state-based (extended SHOP)	★ (interleaving)	A1,A2	★★ (replanning for failures, time outs)
(Fernández-Olivares et al. 2007)	OWL-S	★★ (HPDL)	state-based (SIADEX)	★★ (during planning)		★★ (replanning for failures, time outs)
(Kuter and Golbeck 2009)	OWL-S	×	state-based (extended SHOP2)	× (not discussed)	A4, other	× (not discussed)
(Sohrabi and Mcilraith 2009, Sohrabi and McIlraith 2010)	OWL-S	★★★ (SHOP2,PDDL,LTL)	state-based (extended SHOP2)	★★ (during planning)	A1-A3	× (not discussed)
(Uszok et al. 2004)	OWL-S		plan-based (O-Plan2/I-X/I-Plan)	★ (during planning)	A1	× (not discussed)
(Sirin and Parsia 2004, Sirin et al. 2005)	OWL-S	★★★ (SHOP,DL)	state-based (JSHOP + Pellet)	× (not discussed)		× (not discussed)
(Paik and Maruyama 2007)		★ (SHOP2,CSP)	state-based (JSHOP2 + CSP Solver)	× (not discussed)		× (not discussed)
(Lin et al. 2008)	OWL-S	★★★ (SHOP,DL,PDDL)	state-based (extended JSHOP + Pellet)	× (not discussed)		× (not discussed)

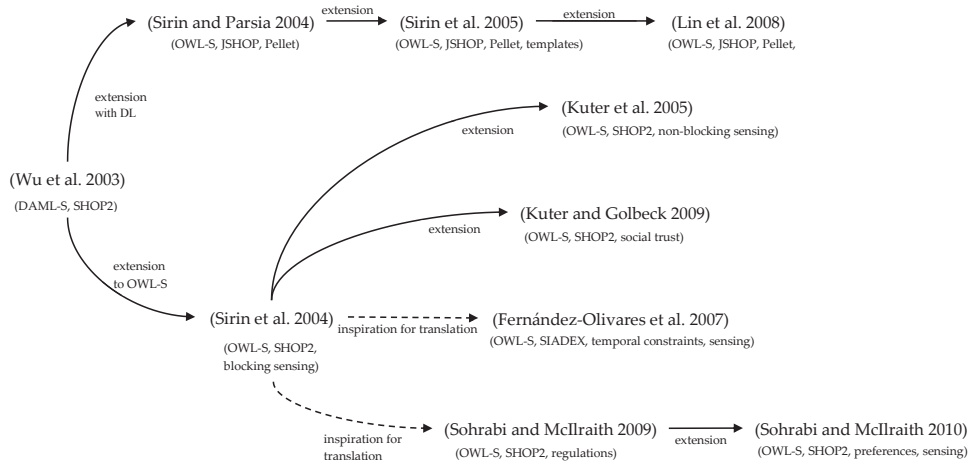


Figure 9.3: Relations between studies that employ HTN planning for Web service composition.

SHOP2. From the plan-based HTN approaches, I-X/I-Plan[9] is employed, which is an HTN planner based on O-Plan2. Most of the approaches give actual contributions to HTN planning by extending the existing algorithms or providing new algorithms on top of the existing planners. With respect to sensing, only a few approaches devote appropriate attention to it and provide a clear description. We can observe and conclude that sensing is done during planning and, in some cases, in a non-blocking manner. While planning, sensing and possibly executing Web services, several approaches make some of the restricting assumptions, at least those that we were able to identify from the descriptions provided. Finally, little attention is devoted to execution monitoring and handling of contingencies at execution time with the exception of the work on using O-Plan2/I-X/I-Plan alongside policy enforcement Web service composition and execution monitoring (Uszok et al. 2004).

Figure 9.3 gives another perspective of approaches that assume OWL-S description of Web services and provide clear translation to the planning-level representation. The lower part specifies the studies that employ HTN planning only. Sirin et al. (Sirin et al. 2004) appears to be the most influential and inspiring study. Two of them are a direct extension of the study, while the other two draw inspiration from the study with respect to the translation process. The upper part depicts the studies that combine HTN planning with DL reasoning. All studies are a continuation of the work presented in the first paper on HTN planning for Web service composition (Wu et al. 2003).

[9] `http://www.aiai.ed.ac.uk/project/i-k-c/`

Chapter 10

Conclusions

With the growth in complexity, computing everywhere is in need of techniques that can advance the computation in an automated, dynamic, and intelligent way. AI planning has the potential to provide such techniques and to bring intelligence everywhere. In this context, our work focused on studying three complementary topics: the field of planning for ubiquitous computing, hierarchical planning as a specific technique relevant for coordination everywhere, and the design and realisation of systems for computing everywhere and the benefits of using them.

10.1 Reflection on planning for ubiquitous computing

Part of our work was set out to explore the field of planning for ubiquitous computing. We first sought to know whether some abstract view of planning for ubiquitous computing can be developed to enhance primarily the understanding of the field. We constructed a model that characterises the concepts constituting the field and the relationships among them. Since the model is grounded in the existing literature, a fit with existing approaches of planning for ubiquitous computing is ensured. While the model provides a consistent way to interpret existing ubiquitous computing systems based on planning, it is also intended to foster more efficient design and development of future systems in ubiquitous computing. Nevertheless, the model can be used as an effective means for communicating with an audience, scientific or non-scientific, that has no prior knowledge in this field.

Another question we set to find an answer to is about what the complexity of solving planning problems in ubiquitous computing is. To that end, we formally defined a general planning domain for ubiquitous computing also grounded in existing literature. Our findings suggest that planning problems in this domain are in **NP** in the worst possible case. While complexity results for other planning domains already exist (*e.g.*, block-worlds (Gupta and Nau 1992) and logistics (Helmert 2003)), to the best of our knowledge, our contribution is first for the domain of ubiquitous computing.

10.2 Reflection on HTN planning

The choice to work with hierarchical planning is made principally due to its rich domain knowledge and the benefits resulting from it. Though practically useful, this long-lived planning technique is associated with controversy and confusion related to issues in both theory and practice. Knowing this, we set out to find answers to the questions of what kind of models of HTN planning exist, which concepts and how they characterise the search space of HTN planners, and what are the properties that describe HTN planning from aspects of domain modelling, expressiveness, competence, computation, and applicability. Our findings indicate that HTN planning can be categorised in plan-based HTN planning and state-based HTN planning, considering the space the search for a solution is performed in. This categorisation differs from the two versions of hierarchical planning discussed by Ghallab et al. (2004) in that our models are distinct styles of HTN planning, each performing search in a different space. In (Ghallab et al. 2004), Simple Task Network (STN) planning is considered as a simplified version of their definition of HTN planning. While our formal models are more specific than the definitions of those two versions, plan-based HTN planning and state-based HTN planning are consistent with their HTN planning and STN planning, respectively.

We constructed a conceptual model demonstrating the concepts that affect the search space and how they are interrelated among each other. Considering the model, we synthesise the following findings:

- Plan-based HTN planners search more complex spaces than state-based HTN planners.
- Plan-based HTN planners have not well-defined task decomposition: Vague information is reported in the literature for nearly all plan-based HTN planners included in our work.
- State-based HTN planners lack non-determinism: In SHOP2 and SIADEX, the task decomposition deterministically chooses a method for the task being currently decomposed.
- Plan-based HTN planners are tightly coupled planning systems: The mechanisms implemented in these planners and used to search, resolve interactions and handle constraints are all highly dependent on each other.
- Plan-based HTN planners make use of various explicit conditions, while state-based HTN planners depend on preconditions: Explicit conditions in plan-based HTN planners support the search process. On the other hand, the whole

reasoning power of state-based HTN planners is encapsulated in the preconditions of both primitive and compound tasks.

With respect to the properties of HTN planners, we synthesise the following:

- Knowing the description language of a plan-based HTN planner is not a sufficient condition to author domain knowledge: To encode knowledge, one needs to first understand planners' underlying mechanisms, such as expectations of what the system would do in a particular situation.
- State-based HTN planners require elaborate domain knowledge: While this is supported by the analysis of results shown in Figure 4.2, additional evidence is the criticism of SHOP2 planner that it is a problem-solving programming language rather than a planner (Schattenberg 2009).
- Rich domain knowledge is a requirement of HTN planners: Though this fact is widely known and accepted in the AI planning community, we supported it by analysing the state-of-the-art HTN planners. Given this, we remark the following observation.

  > "[Compared with classical planners,] the primary advantage of HTN planners is their sophisticated knowledge representation [and reasoning capabilities] (Ghallab et al. 2004)."

  Uncertain is the meaning of "sophisticated". Does it refer to the complexity, richness or some other attribute of the representation? If we assume that it refers to the so-called "knowledge-rich" representation (Wilkins and Desjardins 2001), then the second concern is on HTN planners taking advantage of the use of knowledge-rich encodings. On the one hand, this could be correct, if we consider that these planners improve their performance (over classical planners) thanks to their domain knowledge (Long and Fox 2003). On the other hand, why are HTN planners an advantage if we do not know at what expense, in terms of encoding effort, we obtain that improvement?
- Both categories of HTN planners are able to address a similar level of expressiveness.
- Scarcity of performance evidence: For most of HTN planners, the performance and pairwise comparison are unknown.
- HTN planning is a widely applied planning technique: HTN planning has, so far, been employed in more than 50 applications. More than half of them are tackled with plan-based HTN planning, and SHOP2 is the most applied HTN planner, while O-Plan2 is the most applied plan-based HTN planner.

In addition to clarifying or rectifying some aspects of HTN planning, the exploration of the field provided a perspective on which features can be improved and are not currently addressed. In the context of state-based HTN planning, even though SHOP2 and SIADEX support numerical expressions, the semantics in both planners are left unspecified. Additionally, the model of state-based HTN planning we defined does not include details on numerical expressions either. We therefore sought to develop a formalism that defines these expressions. The resulting model accommodates well-defined numerical expressions in both preconditions and effects of tasks. The model is consistent with HPDL, which is the description language of SIADEX, and based on PDDL. This model also supports the syntax of SHOP2, though it restricts some expressions that are otherwise allowed by the planner, but are less typical for AI planners. To some extent, our model can be seen as a complement to the formalisation developed for SHOP-style of planners (Nau et al. 1999).

The next issue relates to the domain knowledge provided to state-based HTN planners. We showed that these planners require well-conceived knowledge, which, in some cases, such as recursive tasks, involves phantomisation, that is, recognising and dealing with already accomplished facts. While in most plan-based HTN planners such situations are automatically recognised and handled, state-based HTN planners require explicit domain knowledge. We set out to find an answer to the question of how can phantomisation be automated in these planners. We defined the basic notions needed for phantomisation, and extend the algorithm of JSHOP2 to support the phantomisation process. The main implication of the extended planner is that it requires simpler and smaller domain knowledge than what would JSHOP2 need otherwise to solve the same planning problems. On the other hand, the extension of JSHOP2 needs more time to plan than JSHOP2 spends on the same planning problems with explicit knowledge about phantomisation.

Part of our work on state-based HTN planning was set out to explore how tasks help in expressing a certain attitude towards the risk of using a given resource. We defined the framework of utility-based HTN planning in which tasks are associated with utilities that represent some risk attitude. The framework consists of several functions and an algorithm that finds an optimal solution with respect to the resource consumed. Some of the functions are consistent with the ones in (Kuter and Golbeck 2009), where they are used to compute social trust of Web service compositions. This may be considered as a specific case of using utilities in HTN planning.

The ability of AI planners to be easily integrated in larger systems and interoperate with other components of those systems is essential to planners' adoption and use in actual applications. We sought to know how can planners support distribution, evolution and interoperation in an easy and unified way. We proposed the

concept of planning as a service, which is based on the service-orientation principle. This means that the functionalities of planners are designed, implemented and offered as services, respecting the common design principles of service-orientation (Erl 2007). Having planning functionalities as services would provide many of the benefits that Service-Oriented Computing guarantees, such as rapid prototyping, easy addition of new functionalities, scalability, reuse, *etc.* (Papazoglou and Georgakopoulos 2003). Our concept of planning as a service is consistent with the planning services designed for the domain of space missions (Fratini et al. 2013). It also answers the questions related to runtime behaviour, interoperability, and scalability of planners in ubiquitous computing raised but not answered in (Marquardt and Uhrmacher 2009b).

10.3 Reflection on the developed systems

We designed and developed **SH** to be an HTN planning system that accepts a well-defined syntax, and can integrate in a wide range of large and distributed systems. **SH** accepts planning problems specified in HPDL, and offers its functionalities as services. In contrast to JSHOP2, **SH** is a simple and flexible implementation characterised by many of the benefits of service-oriented systems. We used and integrated the planner in a ubiquitous computing system, and employed it in the domain of Cloud applications. On the other hand, we tested and compared **SH** with JSHOP2 in solving planning problems from several domains. The results show that **SH** can perform better or worse than JSHOP2, depending on the evaluated domain.

One of the main objectives of our work was to establish a correspondence between ubiquitous computing and HTN planning, and to employ HTN planning in an actual application. We proposed a correct correspondence between a ubiquitous computing problem and an HTN planning problem. The implication of this is the fact that the plan computed for the HTN planning problem is indeed a solution to the underlying ubiquitous computing problem. Additionally, we enhanced this approach with a feature, called orchestration, to answer the question of how to deal with inconsistencies happening at execution time.

To evaluate the feasibility of the approach, we designed a system architecture and implemented a system prototype. We deployed and used the prototype in the Bernoulliborg restaurant. The results indicate that HTN planning with its rich domain knowledge is a usable and effective technique for automated coordination of ubiquitous services. The effectiveness is demonstrated through energy and monetary savings in a matter of 80%. The feasibility of the approach is also confirmed through evaluations of the usability of the prototype and performance of **SH**. The usability evaluation shows that the majority of the participants find the system use-

ful and effective. The performance evaluation shows that the time required for computing plans in realistic situations is in order of milliseconds, and in extreme cases several seconds can be spent for computation.

Part of our work focused on examining the problem of optimising costs of ubiquitous computing environments connected to the smart grid. We proposed a centralised approach based on scheduling to control appliances in such environments. To evaluate the feasibility of the approach, we designed and realised another system prototype, which we deployed in offices on the fifth floor of Bernoulliborg. The results show that our approach provides an effective way to coordinate the use of devices while balancing the use of power in peak hours. The effectiveness of the prototype is expressed through savings of up to 50% in money and 15% in energy.

Finally, we sought to know how can the process of composing Cloud applications ready for deployment be automated using HTN planning. We established a correspondence between a deployment problem and an HTN planning problem, allowing to demonstrate the completeness of our approach. To evaluate the feasibility of the approach, we encoded domain knowledge in HPDL and used **SH** to solve deployment-based HTN planning problems. We tested the planner on a set of problems and the results show that the solutions produced by **SH** are consistent with those reported in (Lascu et al. 2013). The performance results show that HTN planning with its rich domain knowledge is able to solve deployment problems in a matter of few seconds. The results contradict the proposition that general-purpose planners, such as **SH**, show unsatisfactory performance when composing Cloud applications (Lascu et al. 2013). Considering the results of the planner specifically created for solving this kind of problems presented in the same study, **SH** falls behind the specialised planner only after a reasonably high number of components.

10.4 Limitations

The conceptual model of planning for ubiquitous computing paves the way towards defining the scope and objective of the field, and can facilitate efficient design and development of future ubiquitous computing systems. The model however gives only a broad perspective of the field, leaving out many details and more specific aspects of this phenomenon. The specification of the conceptual model is based on UML classes without any attributes that may characterise better the concepts represented by them. While we used standard UML constructs, some extension of UML specifically designed for constructing conceptual models, such as (Guizzardi 2005), may provide better insights into the concepts and their relationships.

The results on complexity of planning in ubiquitous computing are also general indications, leaving space for a more insightful analysis. Given our general plan-

ning domain for ubiquitous computing, one can further develop more specific domains within ubiquitous computing. For example, the complexity can be analysed with respect to five features: the number of controllables, sources, applications, humans, and robots. Since each of these features can be quantified, the result will be a set of domains with varying difficulty. Consequently, the complexity results may be refined and appear to be different than the one for the general planning domain.

In the context of state-based HTN planning, our framework for utility-based HTN planning should be further extended to support the case of recursive tasks, as a powerful construct in HTNs. In the current framework, we assume that primitive tasks have predefined costs, which are used to calculate utilities using a provided set of utility functions. While this is a practical approach, it may not reflect actual situations. Learning the costs and utilities directly from the environment and attitude of people, may offer a broader and more useful perspective of utility-based HTN planning. The realisation of the framework in a prototype and recognising the actual benefits of its use are yet to be considered.

Considering the **SH** planner, there might be some domains in which the planner will demonstrate less satisfactory performance, as showed in the comparison with JSHOP2. One possibility for improvement of the performance is to consider heuristics that can speed up the search, something in the manner of, for example, (Sohrabi et al. 2008, Alford et al. 2014). Moreover, the plans that **SH** generates can only be totally ordered. While this ordering appears to be sufficient for our case in ubiquitous computing, most of real-world domains require a better quality of plans. If, for example, a Cloud application has a deployment run of partially ordered components, some of the components can be deployed in parallel reducing the deployment time of the application drastically. Furthermore, the services that **SH** offers currently include only basic modelling and problem-solving functionalities. Extending the list of services and possibly having them available in public can make the planner more capable, integrable, and useful.

In the context of HTN planning for ubiquitous computing, we took a pragmatic approach towards dealing with uncertainty during runtime. While this approach was sufficient for the environments of our interest, more capable HTN-based techniques, such as those developed for healthcare domains (Sánchez-Garzón et al. 2011, Fernández-Olivares et al. 2012, Sánchez-Garzón et al. 2013), may prove to be appropriate for ubiquitous computing too. Though we defined basic execution semantics, a provable sound and complete algorithm for execution is still needed.

In reality, the components of Cloud applications have capacity constraints. Though such constraints are not take into account by our approach, it is straightforward to provide an extension that will support this feature. For the coordination of the deployment of application components, we assumed that there is only one

server available for the deployment. However, interesting and real is the case when the coordination is performed with respect to available resources, such as the number of servers, their available space, CPU, *etc.*

10.5 Future directions

To further develop the field of planning for ubiquitous computing, there is a need for more studies on several subjects. More specifically, we find preferences, and spatial and temporal properties insufficiently investigated. Preferences are important for people because they enhance their experience in the environments. Through preferences, people can customise the environments according to their needs and depending on their context, social status, *etc.* User preferences are also challenging because they represent sensitive information that must be considered with care (Bettini and Riboni 2015). The current attention given to preferences is almost non-existing with the exception of (Ranganathan and Campbell 2004).

The most common way of dealing with spatial information in planning for ubiquitous computing is by representing it abstractly. The main issue with the abstract spatial representation is realisibility, that is, locations have to be topologically connected and nothing can travel between locations at infinite speed (Pfender and Ziegler 2004). This can be achieved by purely spatial representation and represents a possibility for future work. In addition, the spatial information about people considered during planning is usually based on spatial generalisation (Mascetti et al. 2014). Spatial generalisation decreases the precision of location information, and in the case of planning approaches, it usually involves the relative location of people. This approach appears to be limiting as it neglects the physical constraints, posture and orientation of humans, which are important in many scenarios in ubiquitous computing (*e.g.,* emergency situations). The main concern of using such information relates to the involved personal data, people's movements, their behavioural habits, and so forth. This implies that the spatial information has to be acquired and reasoned over considering the privacy of people. People need to be aware of the use of their personal data and the manner in which that data is used in ubiquitous computing systems (Bettini and Riboni 2015).

Considering temporal properties, totally ordered plans have appeared to be practically sufficient for the environment we dealt with. However, it would be interesting to observe the practical benefits of partial order, which is identified as useful for plan representation in ubiquitous computing, *e.g.,* (Mastrogiovanni et al. 2010, Bidot et al. 2011, Kaldeli et al. 2012, Pajares Ferrando and Onaindia 2013, Heider 2003, Milani and Poggioni 2007). Additionally, we find limitations in the current use of metric and qualitative relations. Elementary use of metric constraints is re-

ported in (Bajo et al. 2009, Sánchez-Garzón et al. 2012, Fraile et al. 2013), and of qualitative relations in (Rocco et al. 2014).

The typical, manual process of creating domain knowledge is strenuous, becoming even worse and impractical when the planning system is envisioned to be used in more than one ubiquitous computing setting. For the correspondence between HTN planning and ubiquitous computing, we assumed that the environment conditions encoded and maintained in HTNs are derived from standards, policies, *etc.* associated with the respective ubiquitous computing environment. Currently, this is all accomplished manually, leaving space for uncovered situations and error-prone encodings. So, one can benefit, at least practically, from tools that support engineering, automated creation, and learning domain knowledge, such as those reported in (Grześ et al. 2014, Ortiz et al. 2013). In the context of HTN planning, there is a number of systems that use different approaches to learn domain knowledge from examples: one approach learns preconditions of SHOP-like methods given the method structure as input to the system (Ilghami and Nau 2002), another one learns incrementally approximate preconditions (Ilghami et al. 2005), an approach learns very general HTNs by learning from expert traces (Nejati et al. 2006), or another one that learns HTNs with a better balance between generality and specificity (Nejati et al. 2009). Some approaches acquire methods by analysing a set of planning problems together with their solutions and a set of annotated tasks in a given deterministic domain (Hogg et al. 2008), and others may learn knowledge for domains that include primitive tasks with multiple possible outcomes (Hogg et al. 2009). Some recent approaches learn tasks by using a set of partially observed plan traces and a set of annotated tasks together with some constraints (Zhuo et al. 2014), or learn probabilistic HTNs that capture user preferences on plans by observing the user behaviour (Li et al. 2014).

Automatically created planning problems may capture better ubiquitous computing environments assuming that diverse constructs, such as conditional effects, multi-type elements, numeric-valued fluents, *etc.* are available and supported. When the support for an extensive set of such constructs is in question, finding the right balance between the expressiveness and complexity of planning is crucial. Thus, there is space for analysis of the constructs needed to support ubiquitous computing environments and the effect of their use. Related to the representation issues is the one of having a taxonomy for planning in ubiquitous computing. A common taxonomy may provide for consistency in descriptions of proposed approaches, understanding better what others have accomplished, and comparisons of different planning techniques. In this context, our conceptual model and general planning domain can foster the use of standard terms.

New models can be considered for further categorisation of HTN planning, such

as those recently investigated in (Alford et al. 2012). Also, new concepts, *e.g.,* landmarks (Elkawkagy et al. 2010, 2011, 2012), can be plugged into the framework of concepts concerning the search space of planners. We are of the opinion that a common syntax should be used for specifying HTN planning problems. A single language for both categories of planners seems illusory ambitious, but each category can make use of its own language. We contributed to this by choosing to work with HPDL. In this way, research improvements and performance evaluation of HTN planners can be stimulated, a direct comparison of the planners on possibly standardised set of problems can be enabled, and finally, help in understanding the expressive power of HTN planners can be provided. Furthermore, we recognise that HTN planners can be improved in the area of goals, such as extended goals (Lago et al. 2002), hybrid goals (Estlin et al. 2001), and active goal reasoning (Shivashankar et al. 2013).

In our approach of using HTN planning for ubiquitous computing, an HTN planning problem depends on the activities happening in ubiquitous computing environments and derived by specialised activity-recognition techniques. Some of these techniques use rules or conditions to reason for activities. Since HTN planning already has rich domain knowledge, perhaps, instead of using additional rules, HTNs can be used so as to recognise activities, something in the manner of recognising goals in (Pattison and Long 2010). In this way, we may avoid defining and maintaining knowledge at multiple points in systems.

Coordinating devices in offices connected to the smart grid might be possible using HTN planning. The device policies that the scheduler uses represent domain-specific knowledge that can fit into HTNs. Also, since the information coming from energy providers is known in advance for each day, it may be used during planning to choose the one provider that offers the best combination of price and energy.

To summarise, we gave Theodore a real chance to have Samantha as an intelligent system present everywhere. Samantha can now coordinate ubiquitous services automatically and dynamically, but she may also consider the price of energy that Theodore has to pay for coordinated devices. This implies improved quality of Theodore's life in terms of comfort, well-being, and economics. Nevertheless, Theodore can express his requirements and needs for his home, and Samantha can automatically upgrade herself with the necessary Cloud services in no time.

Our work was motivated by the need to have more capable, adaptive, and proactive ubiquitous computing systems, that is, to bring intelligence everywhere. Though Theodore has Samantha to absorb intelligently many of his needs, we wonder, what if he aspires for something like J.A.R.V.I.S. (Just a Rather Very Intelligent System)?[1] Where and how will AI planning fit into systems such as J.A.R.V.I.S.?

[1] J.A.R.V.I.S. is an AI system in the film "Iron Man" (Bettany 2008) and its sequels.

Bibliography

Aarup, M., Arentoft, M. M., Parrod, Y., Stokes, I., Vadon, H. and Stader, J.: 1994, OPTIMUM-AIV: A knowledge-based planning and scheduling system for spacecraft AIV, *Conference on Knowledge Based Scheduling*, pp. 451–469.

Abowd, G. D., Dey, A. K., Brown, P. J., Davies, N., Smith, M. and Steggles, P.: 1999, Towards a better understanding of context and context-awareness, *International Symposium on Handheld and Ubiquitous Computing*, HUC'99, pp. 304–307.

Advantic Sys.: 2015. Online: accessed May 2015, http://www.advanticsys.com/.

Agosta, J. M.: 1996, Constraining influence diagram structure by generative planning: An application to the optimization of oil spill response, *International Conference on Uncertainty in Artificial Intelligence*, UAI'96, pp. 11–19.

Aiello, M., Papazoglou, M. P., Yang, J., Carman, M., Pistore, M., Serafini, L. and Traverso, P.: 2002, A request language for Web services based on planning and constraint satisfaction, *International Workshop on Technologies for E-Services*, Springer, pp. 76–85.

Aiello, M., Pratt-Hartmann, I. and van Benthem, J.: 2007a, *Handbook of spatial logics*, Springer.

Aiello, M., Pratt-Hartmann, I. and Van Benthem, J.: 2007b, What is spatial logic?, *Handbook of spatial logics*, Springer, pp. 1–11.

Alexander, I. and Maiden, N.: 2004, *Scenarios, stories, use cases: Through the systems development life-cycle*, John Wiley & Sons.

Alford, R., Shivashankar, V., Kuter, U. and Nau, D.: 2014, On the feasibility of planning graph style heuristics for HTN planning, *International Conference on Automated Planning and Scheduling*, pp. 2–10.

Alford, R., Shivashankar, V., Kuter, U. and Nau, D. S.: 2012, HTN problem spaces: Structure, algorithms, termination, *Annual Symposium on Combinatorial Search*, pp. 2–9.

Allen, J. F.: 1983, Maintaining knowledge about temporal intervals, *Comm. ACM* **26**(11), 832–843.

Amigoni, F., Gatti, N., Pinciroli, C. and Roveri, M.: 2005, What planner for ambient intelligence applications?, *IEEE Transactions on Systems, Man and Cybernetics, Part A* **35**(1), 7–21.

Andréka, H., Madarász, J. X. and Németi, I.: 2007, Logic of space-time and relativity theory, *Handbook of spatial logics*, Springer, pp. 607–711.

Andrews, S., Kettler, B., Erol, K. and Hendler, J.: 1995, UM Translog: A planning domain for the development and benchmarking of planning systems, *Technical Report CS-TR-3487*, Computer Science Department, University of Maryland.

Arshad, N., Heimbigner, D. and Wolf, A.: 2003, Deployment and dynamic reconfiguration planning for distributed software systems, *International Conference on Tools with Artificial Intelligence*, ICTAI'03, pp. 39–46.

Asunción, M. d. l., Castillo, L., Fernádez-Olivares, J., García-Pérez, O., González, A. and Palao, F.: 2005, SIADEX: An interactive artificial intelligence planner for decision support and training in forest fire fighting, *Artificial Intelligence Communications* **18**(4), 257–268.

Bacchus, F.: 2001, The AIPS '00 Planning Competition, *AI Magazine* **22**(3), 47–56.

Bacchus, F. and Kabanza, F.: 1996, Using temporal logic to control search in a forward chaining planner, *in* M. Ghallab and A. Milani (eds), *New directions in AI planning*, IOS Press, pp. 141–153.

Bacchus, F. and Kabanza, F.: 2000, Using temporal logics to express search control knowledge for planning, *Artif. Intell.* **116**(1–2), 123–191.

Bäckström, C. and Nebel, B.: 1995, Complexity results for SAS+ planning, *Computational Intelligence* **11**(2-3), 625–655.

Bajo, J., de Paz, J. F., de Paz, Y. and Corchado, J. M.: 2009, Integrating case-based planning and RPTW neural networks to construct an intelligent environment for health care, *Expert Syst. Appl.* **36**(3), 5844–5858.

Bauer, C. and King, G.: 2006, *Java persistence with hibernate*, Manning Pubs Co.

Benthem, J. F. A. K. v.: 1991, *The logic of time: A model-theoretic investigation into the varieties of temporal ontology and temporal discourse*, 2nd edn, Kluwer Academic Publishers.

Benthem, J. v.: 1983, *The logic of time: A model-theoretic investigation into the varieties of temporal ontology and temporal discourse*, Springer.

Benton, J., Coles, A. J. and Coles, A.: 2012, Temporal planning with preferences and time-dependent continuous costs, *International Conference on Automated Planning and Scheduling*, pp. 2–10.

Berardi, D., Cheikh, F., Giacomo, G. D. and Patrizi, F.: 2008, Automatic service composition via simulation, *International Journal of Foundations of Computer Science* **19**(2), 429–451.

Bettany, P.: 2008, *Iron Man*, Directed by Jon Favreau. Marvel Studios, Fairview Entertainment, USA. Film.

Bettini, C., Brdiczka, O., Henricksen, K., Indulska, J., Nicklas, D., Ranganathan, A. and Riboni, D.: 2010, A survey of context modelling and reasoning techniques, *Pervasive Mob. Comput.* **6**(2), 161–180.

Bettini, C. and Riboni, D.: 2015, Privacy protection in pervasive systems: State of the art and technical challenges, *Pervasive Mob. Comput.* **17, Part B**(0), 159–174.

Bidot, J., Goumopoulos, C. and Calemis, I.: 2011, Using AI planning and late binding for managing service workflows in intelligent environments, *International Conference on Pervasive Computing and Communications*, PERCOM '11, IEEE, pp. 156–163.

Blaylock, N. and Allen, J.: 2005, Generating artificial corpora for plan recognition, *International Conference on User Modeling*, UM'05, Springer, pp. 179–188.

Blaylock, N. and Allen, J.: 2006, Hierarchical instantiated goal recognition, *AAAI Workshop on Modeling Others from Observations*, pp. 8–15.

Bonet, B. and Geffner, H.: 2000, Planning with incomplete information as heuristic search in belief space, *International Conference on Artificial Intelligence Planning and Scheduling*.

Booch, G., Rumbaugh, J. and Jacobson, I.: 2005, *The unified modeling language user guide, 2nd edition*, Addison-Wesley Professional.

Bratko, I.: 2001, *Prolog (3rd Ed.): Programming for Artificial Intelligence*, Addison-Wesley Longman Publishing Co., Inc.

Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E. and Yergeau, F.: 2008, Extensible markup language (XML) 1.0 (Fifth edition). Online: accessed Sep 2014, http://www.w3.org/TR/REC-xml/.

Brenner, M. and Nebel, B.: 2009, Continual planning and acting in dynamic multiagent environments, *Autonomous Agents and Multi-Agent Systems* **19**(3), 297–331.

Brnsted, J., Hansen, K. M. and Ingstrup, M.: 2010, Service composition issues in pervasive computing, *Pervasive Computing, IEEE* **9**(1), 62–70.

Bulterman, D. C. A.: 2001, Smil 2.0 part 1: Overview, concepts, and structure, *MultiMedia, IEEE* **8**(4), 82–88.

Butz, A.: 2010, User interfaces and HCI for ambient intelligence and smart environments, *Handbook of Ambient Intelligence and Smart Environments*, Springer, pp. 535–558.

Bylander, T.: 1994, The computational complexity of propositional STRIPS planning, *Artif. Intell.* **69**(1-2), 165–204.

Carolis, B. and Cozzolongo, G.: 2007, Planning the behaviour of a social robot acting as a majordomo in public environments, *Congress of the Italian Association for Artificial Intelligence on AI*IA 2007: Artificial Intelligence and Human-Oriented Computing*, pp. 805–812.

Casati, R. and Varzi, A. C.: 1999, *Parts and places: The structures of spatial representation*, MIT Press.

Castillo, L. A., Fernández-Olivares, J., García-Pérez, Ó. and Palao, F.: 2005, Temporal enhancements of an HTN planner, *Conference of the Spanish Association for Artificial Intelligence, Current Topics in AI*, pp. 429–438.

Castillo, L. A., Fernández-Olivares, J., García-Pérez, Ó. and Palao, F.: 2006, Efficiently handling temporal knowledge in an HTN planner, *International Conference on Automated Planning and Scheduling*, ICAPS'06, AAAI, pp. 63–72.

Castillo, L., Armengol, E., Onaindía, E., Sebastiá, L., González-Boticario, J., Rodríguez, A., Fernández, S., Arias, J. D. and Borrajo, D.: 2008, samap: An user-oriented adaptive system for planning tourist visits, *Expert Syst. Appl.* **34**(2), 1318–1332.

Castillo, L., Morales, L., González-Ferrer, A., Fernández-Olivares, J., Borrajo, D. and Onaindía, E.: 2010, Automatic generation of temporal planning domains for e-learning problems, *J. of Scheduling* **13**(4), 347–362.

Chenoweth, S. V.: 1991, On the NP-hardness of blocks world, *National Conference on Artificial Intelligence*, AAAI'91, pp. 623–627.

Cirillo, M., Karlsson, L. and Saffiotti, A.: 2012, Human-aware planning for robots embedded in ambient ecologies, *Pervasive Mob. Comput.* **8**(4), 542–561.

ConsortiumW3C, W. W. W.: 2005, Web service semantics: WSDL-S.

Cook, D. and Das, S.: 2004, *Smart environments: Technology, protocols and applications (Wiley Series on Parallel and Distributed Computing)*, Wiley-Interscience.

Corbin, J. M. and Strauss, A.: 1990, Grounded theory research: Procedures, canons, and evaluative criteria, *Qualitative Sociology* **13**(1), 3–21.

Corbin, J. and Strauss, A.: 2008, *Basics of qualitative research: Techniques and procedures for developing grounded theory*, Sage.

Cosmo, R. d., Zacchiroli, S. and Zavattaro, G.: 2012, Towards a formal component model for the cloud, *International Conference on Software Engineering and Formal Methods*, SEFM'12, pp. 156–171.

Courtemanche, F., Najjar, M., Paccoud, B. and Mayers, A.: 2008, Assisting elders via dynamic multi-tasks planning: A markov decision processes based approach, *International Conference on Ambient Media and Systems*, pp. 1–8.

Currie, K. and Tate, A.: 1991, O-Plan: The open planning architecture, *Artif. Intell.* **52**(1), 49–86.

Curry, M. R.: 1996, *The work in the world - Geographical practice and the written word*, University of Minnesota Press.

Degeler, V. and Lazovik, A.: 2013, Dynamic constraint reasoning in smart environments, *IEEE International Conference on Tools with Artificial Intelligence*, ICTAI.

Degeler, V., Lopera Gonzalez, L. I., Leva, M., Shrubsole, P., Bonomi, S., Amft, O. and Lazovik, A.: 2013, Service-oriented architecture for smart environments, *IEEE International Conference on Service Oriented Computing and Applications*, SOCA'13, pp. 99–104.

Ding, Y., Elting, C. and Scholz, U.: 2006, Seamless integration of output devices in intelligent environments: Infrastructure, strategies and implemeentation, *IET International Conference on Intelligent Environments*, pp. 21–30.

Do, M. B. and Kambhampati, S.: 2003, Sapa: A multi-objective metric temporal planner, *J. Artif. Int. Res.* **20**(1), 155–194.

Drabble, B., Dalton, J. and Tate, A.: 1997, Repairing plans on-the-fly, *NASA Workshop on P&S for Space*.

Drummond, M. E., Currie, K. W. and Tate, A.: 1988, O-Plan meets T-SAT: First results from the application of an AI planner to spacecraft mission sequencing, *Technical report*, Artificial Intelligence Applications Institute, University of Edinburgh.

Dustdar, S. and Schreiner, W.: 2005, A survey on Web services composition, *Int. J. Web Grid Serv.* **1**(1), 1–30.

Edelkamp, S. and Helmert, M.: 2001, MIPS: the model-checking integrated planning system, *AI Magazine* **22**(3), 67–72.

Elkawkagy, M., Bercher, P., Schattenberg, B. and Biundo, S.: 2011, Landmark-aware strategies for hierarchical planning, *Workshop on Heuristics for Domain-independent Planning (HDIP 2011) at ICAPS 2011*, pp. 73–79.

Elkawkagy, M., Bercher, P., Schattenberg, B. and Biundo, S.: 2012, Improving hierarchical planning performance by the use of landmarks, *AAAI Conference on Artificial Intelligence*, pp. 1763–1769.

Elkawkagy, M., Schattenberg, B. and Biundo, S.: 2010, Landmarks in hierarchical planning, *European Conference on Artificial Intelligence*, IOS Press, pp. 229–234.

Embley, D. W. and Thalheim, B.: 2011, *Handbook of conceptual modeling: Theory, practice, and research challenges*, Springer.

Erl, T.: 2007, *SOA principles of service design*, Prentice Hall PTR.

Erol, K.: 1996, *Hierarchical task network planning: Formalization, analysis, and implementation*, PhD thesis, Computer Science Department, University of Maryland.

Erol, K., Handler, J. and Nau, D. S.: 1996, Complexity results for HTN planning, *Annals of Mathematics and AI* **18**(1), 69–93.

Erol, K., Hendler, J. and Nau, D. S.: 1994a, HTN planning: Complexity and expressivity, *National Conference on Artificial Intelligence - Volume 2*, AAAI, pp. 1123–1128.

Erol, K., Hendler, J. and Nau, D. S.: 1994b, Semantics for hierarchical task network planning, *Technical Report UMIACS-TR-94-31*, University of Maryland, Institute for Advanced Computer Studies.

Erol, K., Hendler, J. and Nau, D. S.: 1994c, UMCP: A sound and complete procedure for hierarchical task network planning, *International Conference on AI Planning Systems*, pp. 249–254.

Erol, K., Nau, D. S. and Subrahmanian, V. S.: 1995, Complexity, decidability and undecidability results for domain-independent planning, *Artif. Intell.* **76**(1-2), 75–88.

Estlin, T. A., Chien, S. A. and Wang, X.: 2001, Hierarchical task network and operator-based planning: Two complementary approaches to real-world planning, *Journal of Experimental and Theoretical Artificial Intelligence* **13**(4), 379–395.

European Committee for Standardization: 2011, Light and lighting - Lighting of work places - Part 1: Indoor work places, *European standard*, Official Journal of the European Union.

Fan, J. and Kambhampati, S.: 2005, A snapshot of public web services, *SIGMOD Rec.* **34**(1), 24–32.

Fernández-Olivares, J., Castillo, L., García-Pérez, O. and Palao, F.: 2006, Bringing users and planning technology together. Experiences in SIADEX, *International Conference on Automated Planning and Scheduling*, ICAPS'06, pp. 11–20.

Fernández-Olivares, J., Cózar, J. A. and Castillo, L.: 2008, Automating oncology therapy plans by means of temporal hierarchical task networks planning, *ECAI Workshop on Knowledge Management for Healthcare Processes*.

Fernández-Olivares, J., Garzón, T., Castillo, L., García-Pérez, O. and Palao, F.: 2007, A middle-ware for the automated composition and invocation of Semantic Web services based on temporal HTN planning techniques, *in* D. Borrajo, L. Castillo and J. Corchado (eds), *Current Topics in Artificial Intelligence*, Vol. 4788 of *Lecture Notes in Computer Science*, Springer, pp. 70–79.

Fernández-Olivares, J., Sánchez-Garzón, I., González-Ferrer, A., Cózar, J. A., Fdez-Teijeiro, A., Cabello, M. R. and Castillo, L.: 2012, Task network based modeling, dynamic generation and adaptive execution of patient-tailored treatment plans based on smart process management technologies, *International Conference on Knowledge Representation for Health-Care*, KR4HC'11, pp. 37–50.

Fielding, R. T. and Taylor, R. N.: 2002, Principled design of the modern web architecture, *ACM Trans. Internet Technol.* **2**(2), 115–150.

Fikes, R. E. and Nilsson, N. J.: 1971, STRIPS: A new approach to the application of theorem proving to problem solving, *International Joint Conference on Artificial Intelligence*, IJCAI'71, pp. 608–620.

Foulser, D. E., Li, M. and Yang, Q.: 1992, Theory and algorithms for plan merging, *Artif. Intell.* **57**, 143–181.

Fox, M. and Long, D.: 2003, PDDL2.1: An extension to PDDL for expressing temporal planning domains, *Journal of Artificial Intelligence Research* **20**(1), 61–124.

Fraile, J. A., Paz, Y., Bajo, J., Paz, J. F. and Pérez-Lancho, B.: 2013, Context-aware multiagent system: Planning home care tasks, *Knowledge and Information Systems* pp. 1–33.

Fratini, S., Policella, N. and Donati, A.: 2013, A service oriented approach for the interoperability of space mission planning systems, *Workshop on Knowledge Engineering for Planning and Scheduling*, pp. 39–43.

Gancet, J., Hattenberger, G., Alami, R. and Lacroix, S.: 2005, Task planning and control for a multi-UAV system: architecture and algorithms, *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 1017–1022.

Garey, M. R. and Johnson, D. S.: 1990, *Computers and intractability; A guide to the theory of NP-completeness*, W. H. Freeman & Co.

Garro, A., Greco, S. and Palopoli, F.: 2008, Smart agents and smart environments: A predictive approach to replanning, *Intelligent Agents and Services for Smart Environments (IASSE) as part of the Artificial Intelligence and Simulation of Behaviour (AISB) Convention*, pp. 7–12.

Geier, T. and Bercher, P.: 2011, On the decidability of HTN planning with task insertion, *International Joint Conference on Artificial Intelligence - Volume 3*, IJCAI'11, AAAI, pp. 1955–1961.

Georgievski, I.: 2013, HPDL: Hierarchical Planning Definition Language, *JBI Preprint 2013-12-3*, Uni. of Groningen.

Georgievski, I. and Aiello, M.: 2014, An overview of hierarchical task planning, *Technical report*, CoRR, abs/1403.7426.

Georgievski, I. and Aiello, M.: 2015a, HTN planning: Overview, comparison, and beyond, *Artif. Intell.* **222**, 124–156.

Georgievski, I. and Aiello, M.: 2015b, Planning in ubiquitous computing: Classes, model, and complexity, *JBI Preprint 2013-12-2*, Uni. of Groningen.

Georgievski, I., Degeler, V., Pagani, G. A., Nguyen, T. A., Lazovik, A. and Aiello, M.: 2012, Optimizing energy costs for offices connected to the smart grid, *IEEE Transactions on Smart Grid* **3**, 2273–2285.

Georgievski, I. and Lazovik, A.: 2014, Utility-based HTN planning, *European Conference on Artificial Intelligence*, IOSPress, pp. 1013–1014.

Georgievski, I., Lazovik, A. and Aiello, M.: 2011, Task interaction in an HTN planner, *Technical report*, CoRR, abs/1111.7025.

Georgievski, I., Nguyen, T. A. and Aiello, M.: 2013, Combining activity recognition and ai planning for energy-saving offices, *International Conference on Ubiquitous Intelligence and Computing*, IEEE, pp. 238–245.

Gerevini, A. and Long, D.: 2006, Preferences and soft constraints in PDDL3, *ICAPS Workshop on Planning with Preferences and Soft Constraints*.

Gerevini, A. and Saetti, A.: 2008, An interactive environment for plan visualization and generation: InLPG, *International Conference on Automated Planning and Scheduling (System Demo)*.

Gerevini, A., Saetti, A. and Serina, I.: 2006, An approach to temporal planning and scheduling in domains with predictable exogenous events, *J. Artif. Int. Res.* **25**(1), 187–231.

Gerevini, A. and Serina, I.: 2002, LPG: A planner based on local search for planning graphs with action costs, *International Conference on Artificial Intelligence Planning Systems*, pp. 13–22.

Ghallab, M., Nau, D. S. and Traverso, P.: 2004, *Automated planning: Theory & practice*, Morgan Kaufmann Publishers Inc.

Glaser, B. G. and Strauss, A. L.: 2009, *The discovery of grounded theory: Strategies for qualitative research*, Aldine de Gruyter.

Goldman, R. P.: 2006, Durative planning in HTNs, *International Conference on Automated Planning and Scheduling*, ICAPS'06, pp. 382–385.

González-Ferrer, A., Fernández-Olivares, J. and Castillo, L.: 2013, From business process models to hierarchical task network planning domains, *Knowledge Eng. Review* **28**(2), 175–193.

Grześ, M., Hoey, J., Khan, S. S., Mihailidis, A., Czarnuch, S., Jackson, D. and Monk, A.: 2014, Relational approach to knowledge engineering for pomdp-based assistance systems as a translation of a psychological model, *International Journal of Approximate Reasoning* **55**(1, Part 1), 36–58.

Guesgen, H. W. and Marsland, S.: 2010, Spatio-temporal reasoning and context awareness, *in* H. Nakashima, H. Aghajan and J. Augusto (eds), *Handbook of Ambient Intelligence and Smart Environments*, Springer US, pp. 609–634.

Guizzardi, G.: 2005, *Ontological foundations for structural conceptual models*, PhD thesis, Centre for Telematics and Information Technology, University of Twente.

Gupta, N. and Nau, D. S.: 1992, On the complexity of blocks-world planning, *Artif. Intell.* **56**(2-3), 223–254.

Ha, Y.-G., Sohn, J.-C., Cho, Y.-J. and Yoon, H.: 2005, Towards a ubiquitous robotic companion: Design and implementation of ubiquitous robotic service framework, *Electronics and Telecommunications Research Institute Journal* **27**(6), 666–676.

Hammond, K. J.: 1989, *Case-based planning: Viewing planning as a memory task*, Academic Press Professional, Inc.

Harrington, A. and Cahill, V.: 2011, Model-driven engineering of planning and optimisation algorithms for pervasive computing environments, *Pervasive Mob. Comput.* **7**(6), 705–726.

Hayes, B.: 2008, Cloud computing, *Commun. ACM* **51**(7), 9–11.

Heider, T.: 2003, Goal-oriented assistance for extended multimedia systems and dynamic technical infrastructures, *IASTED International Conference on Internet and Multimedia Systems and Applications*.

Heider, T. and Kirste, T.: 2002, Supporting goal-based interaction with dynamic intelligent environments, *European Conference on Artificial Intelligence*, ECAI'02, pp. 596–600.

Helmert, M.: 2003, Complexity results for standard benchmark domains in planning, *Artif. Intell.* **143**(2), 219–262.

Helmert, M.: 2009, Concise finite-domain representations for PDDL planning tasks, *Artif. Intell.* **173**(5-6), 503–535.

Hewitt, C., Bishop, P. and Steiger, R.: 1973, A universal modular ACTOR formalism for Artificial Intelligence, *International Joint Conference on Artificial Intelligence*, IJCAI'73, pp. 235–245.

Hidalgo, E., Castillo, L., Madrid, R. I., García-Pérez, O., Cabello, M. and Fdez-Olivares, J.: 2011, ATHENA: Smart process management for daily activity planning for cognitive impairment, *in* J. Bravo, R. Hervás and V. Villarreal (eds), *Ambient Assisted Living*, Vol. 6693 of *Lecture Notes in Computer Science*, Springer, pp. 65–72.

Hoekstra, M.: 2013, *Web-based interface for domain manipulation in smart offices*, Bachelor's thesis, University of Groningen.

Hoffmann, J.: 2002, Extending FF to numerical state variables, *European Conference on Artificial Intelligence*, pp. 571–575.

Hoffmann, J.: 2003, The Metric-FF planning system: Translating "ignoring delete lists" to numeric state variables, *J. Artif. Int. Res.* **20**(1), 291–341.

Hoffmann, J. and Brafman, R. I.: 2006, Conformant planning via heuristic forward search: A new approach, *Artif. Intell.* **170**(6–7), 507–541.

Hoffmann, J., Edelkamp, S., Thiébaux, S., Englert, R., dos Santos Liporace, F. and Trüg, S.: 2006, Engineering benchmarks for planning: The domains used in the deterministic part of ipc-4, *J. Artif. Int. Res.* **26**(1), 453–541.

Hoffmann, J. and Nebel, B.: 2001, The FF planning system: Fast plan generation through heuristic search, *J. Artif. Int. Res.* **14**(1), 253–302.

Hogg, C., Kuter, U. and Muñoz Avila, H.: 2009, Learning hierarchical task networks for non-deterministic planning domains, *International Joint Conference on Artifical Intelligence*, IJCAI'09, pp. 1708–1714.

Hogg, C., Muñoz Avila, H. and Kuter, U.: 2008, HTN-MAKER: Learning HTNs with minimal additional knowledge engineering required, *National Conference on Artificial Intelligence - Volume 2*, AAAI, pp. 950–956.

Horrocks, I., Patel-Schneider, P. F. and van Harmelen, F.: 2003, From SHIQ and RDF to OWL: The making of a Web ontology language, *Web Semantics: Science, Services and Agents on the World Wide Web* **1**(1), 7–26.

Iivari, J. and Venable, J. R.: 2009, Action research and design science research - Seemingly similar but decisively dissimilar, *European Conference on Information Systems*.

Ilghami, O.: 2006, Documentation for JSHOP2, *Technical report*, Department of Computer Science, University of Maryland.

Ilghami, O. and Nau, D. S.: 2002, CaMeL: Learning method preconditions for HTN planning, *International Conference on AI Planning and Scheduling*, pp. 131–141.

Ilghami, O., Nau, D. S. and Aha, D. W.: 2005, Learning preconditions for planning from plan traces and HTN structure, *Computational Intelligence* **21**(4), 388–413.

Irwin, B.: 2014, *Interstellar*, Directed by Chirstopher Nolan. Legendary Pictures, Syncopy, Lynda Obst Productions, UK and USA. Film.

Jeusfeld, M. A., Jarke, M. and Mylopoulos, J.: 2009, *Metamodeling for method engineering*, The MIT Press.

Jih, W., Chen, L. and Hsu, J. Y.: 2007, A context-aware service platform in a smart space, *ACM International Workshop on Agent-Based Ubiquitous Computing*.

Jih, W., Hsu, J. Y., Lee, T. and Chen, L.: 2007, A multi-agent context-aware service platform in a smart space, *Journal of Computers* **18**(1), 45–60.

Juve, G. and Deelman, E.: 2011, Automating application deployment in infrastructure clouds, *International Conference on Cloud computing technology and science*, CloudCom'11, IEEE, pp. 658–665.

Kaldeli, E.: 2013, *Domain-independent planning for services in uncertain and dynamic environments*, PhD thesis, Faculty of Mathematics and Natural Sciences, University of Groningen.

Kaldeli, E., Lazovik, A. and Aiello, M.: 2009, Extended goals for composing services, *International Conference on Automated Planning and Scheduling*, ICAPS'09, pp. 362–365.

Kaldeli, E., Lazovik, A. and Aiello, M.: 2011, Continual planning with sensing for Web service composition, *AAAI Conference on Artificial Intelligence*, pp. 1198–1203.

Kaldeli, E., Warriach, E. U., Lazovik, A. and Aiello, M.: 2012, Coordinating the web of services for a smart home, *ACM Transactions on the Web* **7**(2).

Kambhampati, S.: 1995, A comparative analysis of partial order planning and task reduction planning, *SIGART Bull.* **6**, 16–25.

Karlsson, L.: 2001, Conditional progressive planning under encertainty, *International Joint Conference on Artificial Intelligence - Volume 1*, IJCAI'01, pp. 431–436.

Kautz, H. and Selman, B.: 1999, Unifying SAT-based and graph-based planning, *International Joint Conference on Artifical Intelligence - Volume 1*, IJCAI'99, Morgan Kaufmann Publishers Inc., pp. 318–325.

Kelly, J., Botea, A. and Koenig, S.: 2008, Offline planning with hierarchical task networks in video games, *Artificial Intelligence and Interactive Digital Entertainment Conference*.

Khan, S., Gillis, W., Schmidt, C. and Decker, K.: 2003, A multi-agent system-driven AI planning approach to biological pathway discovery, *International Conference on Automated Planning and Scheduling)*, ICAPS'03, pp. 246–255.

Kim, S. H., Kim, S. W. and Park, H.: 2003, Usability challenges in ubicomp environment, *International Ergonomics Association*.

Kingston, J., Shadbolt, N. and Tate, A.: 1996, CommonKADS models for knowledge based planning, *National Conference on Artifical Intelligence*, AAAI, pp. 11–6.

Kirschnick, J., Alcaraz Calero, J. M., Goldsack, P., Farrell, A., Guijarro, J., Loughran, S., Edwards, N. and Wilcock, L.: 2012, Towards an architecture for deploying elastic services in the cloud, *Softw. Pract. Exper.* **42**(4), 395–408.

Kitchenham, B. and Charters, S.: 2007, Guidelines for performing systematic literature reviews in software engineering, *Technical Report EBSE 2007-001*, Keele University and Durham University Joint Report.

Klassen, T. P., Jadad, A. R. and Moher, D.: 1998, Guides for reading and interpreting systematic reviews: I. getting started, *Archives of Pediatrics & Adolescent Medicine* **152**(7), 700–704.

Klusch, M. and Gerber, A.: 2005, Semantic Web service composition planning with OWLS-XPlan, *International AAAI Fall Symposium on Agents and the Semantic Web*, pp. 55–62.

Koenig, S. and Simmons, R. G.: 1994, Risk-sensitive planning with probabilistic decision graphs, *International Conference on Principles of Knowledge Representation and Reasoning*, pp. 363–373.

Kok, K., Derzsi, Z., Jaap, G., Hommelberg, M., Warmer, C., Kamphuis, R. and Akkermans, H.: 2008, Agent-based electricity balancing with distributed energy resources: A multiperspective case study, *Hawaii International Conference on System Sciences*, pp. 173–173.

Kontchakov, R., Pratt-Hartmann, I. and Zakharyaschev, M.: 2014, Spatial reasoning with RCC8 and connectedness constraints in euclidean spaces, *Artif. Intell.* **217**, 43–75.

Kotsovinos, E. and Vukovic, M.: 2005, su-chef: Adaptive coordination of intelligent home environments, *Joint International Conference on Autonomic and Autonomous Systems and International Conference on Networking and Services*, IEEE, pp. 74–74.

Kremer, U., Hicks, J. and Rehg, J.: 2003, A compilation framework for power and energy management on mobile computers, *in* H. Dietz (ed.), *Languages and Compilers for Parallel Computing*, Vol. 2624 of *Lecture Notes in Computer Science*, Springer, pp. 115–131.

Krüger, F., Ruscher, G., Bader, S. and Kirste, T.: 2011, A context-aware proactive controller for smart environments, *I-COM* **10**, 41–48.

Kuter, U. and Golbeck, J.: 2009, Semantic Web service composition in social environments, *International Semantic Web Conference*, ISWC '09, Springer, pp. 344–358.

Kuter, U., Sirin, E., Parsia, B., Nau, D. and Hendler, J.: 2005, Information gathering during planning for Web service composition, *Web Semantic* **3**, 183–205.

Lago, U. D., Pistore, M. and Traverso, P.: 2002, Planning with a language for extended goals, *AAAI/IAAI*, pp. 447–454.

Larman, C.: 2004, *Applying UML and patterns: An introduction to object-oriented analysis and design and iterative development*, 3 edn, Prentice Hall PTR.

Lascu, T. A., Mauro, J. and Zavattaro, G.: 2013, A planning tool supporting the deployment of cloud applications, *International Conference on Tools with Artificial Intelligence*, ICTAI'13, pp. 213–220.

Lazovik, A.: 2006, *Interacting with service compositions*, PhD thesis, International Doctorate School in Information and Communication Technologies, Trento Univ.

Lazovik, A., Aiello, M. and Papazoglou, M.: 2003, Planning and monitoring the execution of web service requests, *in* M. E. Orlowska, S. Weerawarana, M. P. Papazoglou and J. Yang (eds), *Service-Oriented Computing - ICSOC 2003*, Vol. 2910 of *Lecture Notes in Computer Science*, Springer, pp. 335–350.

Lazovik, A., Aiello, M. and Papazoglou, M.: 2004, Associating assertions with business processes and monitoring their execution, *International Conference on Service-Oriented Computing*, ICSOC '04, ACM, pp. 94–104.

Lee, T. J. and Wilkins, D.: 1996, Using SIPE-2 to integrate planning for military air campaigns, *IEEE Expert* **11**(6), 11–12.

Lekavý, M. and Návrat, P.: 2007, Expressivity of STRIPS-like and HTN-like planning, *KES International Symposium on Agent and Multi-Agent Systems: Technologies and Applications*, KES-AMSTA'07, Springer, pp. 121–130.

Lemon, O. and Pratt, I.: 1997, Spatial logic and the complexity of diagrammatic reasoning, *Machine Graphics and Vision* **6**(1), 89–108.

Li, N., Cushing, W., Kambhampati, S. and Yoon, S.: 2014, Learning probabilistic hierarchical task networks as probabilistic context-free grammars to capture user preferences, *ACM Trans. Intell. Syst. Technol.* **5**(2), 29:1–29:32.

Liang, H., Liu, A., Chen, Y. and Leon Lee, C.: 2010, Device collaboration in smarthomes as service delivery, *SICE Annual Conference*, pp. 30–34.

Likert, R.: 1932, A technique for the measurement of attitudes, *Archives of Psychology* **22**(140), 1–55.

Lin, N., Kuter, U. and Sirin, E.: 2008, Web service composition with user preferences, *European Semantic Web Conference on The Semantic Web: Research and Applications*, ESWC'08, pp. 629–643.

Long, D. and Fox, M.: 2003, The 3rd International Planning Competition: Results and analysis, *J. Artif. Int. Res.* **20**, 1–59.

Long, D., Fox, M. and Howey, R.: 2009, Planning domains and plans: Validation, verification and analysis, *ICAPS'09 Workshop on Verification and Validation of P&S Systems*.

Lu, G., De, D. and Song, W.-Z.: 2010, SmartGridLab: A laboratory-based smart grid testbed, *International Conference on Smart Grid Communications*, IEEE, pp. 143–148.

Luo, J., Zhu, C., Zhang, W. and Liu, Z.: 2013, Messy genetic algorithm for optimum solution search of HTN planning, *JICS* **10**(5), 1303–1313.

Madhusudan, T. and Uttamsingh, N.: 2006, A declarative approach to composing Web services in dynamic environments, *Decis. Support Syst.* **41**(2), 325–357.

Madkour, M., El Ghanami, D. and Maach, A.: 2013, Context-aware service adaptation: An approach based on fuzzy sets and service composition, *J. Inf. Sci. Eng.* **29**(1), 1–16.

Marco, D. D., Janssen, R., Perzylo, A., Van de Molengraft, M. J. and Levi, P.: 2013, A deliberation layer for instantiating robot execution plans from abstract task descriptions, *International Conference on Automated Planning and Scheduling (Workshop on Planning and Robotics)*, pp. 12–19.

Marquardt, F., Reisse, C., Uhrmacher, A. and Kirste, T.: 2008, A two-way approach to service composition in smart device ensembles, *Advanced Topics in Telecommunication*, pp. 49–60.

Marquardt, F. and Uhrmacher, A.: 2009a, Creating AI planning domains for smart environments using PDDL, *in* D. Tavangarian, T. Kirste, D. Timmermann, U. Lucke and D. Versick (eds), *Intelligent Interactive Assistance and Mobile Multimedia Computing*, Vol. 53 of *Communications in Computer and Information Science*, Springer, pp. 263–274.

Marquardt, F. and Uhrmacher, A. M.: 2009b, An ai-planning based service composition architecture for ambient intelligence., *Intelligent Environments (Workshops)*, Vol. 4 of *Ambient Intelligence and Smart Environments*, pp. 145–152.

Marquardt, F. and Uhrmacher, A. M.: 2009c, An AI-planning based service composition architecture for ambient intelligence, *Workshop of the International Conference on Intelligent Environments*, pp. 145–152.

Martin, D., Burstein, M., Mcdermott, D., Mcilraith, S., Paolucci, M., Sycara, K., Mcguinness, D. L., Sirin, E. and Srinivasan, N.: 2007, Bringing semantics to Web services with OWL-S, *World Wide Web* **10**(3), 243–277.

Mascetti, S., Bertolaja, L. and Bettini, C.: 2014, Safebox: Adaptable spatio-temporal generalization for location privacy protection, *Transactions on Data Privacy* **7**(2), 131–163.

Masellis, R. D., Ciccio, C. D., Mecella, M. and Patrizi, F.: 2010, Smart home planning programs, *International Conference on Service Systems and Service Management*, ICSSSM, pp. 1–6.

Mastrogiovanni, F., Scalmato, A., Sgorbissa, A. and Zaccaria, R.: 2010, Affordance-based planning for assisting humans in daily activities, *International Conference on Intelligent Environments*, pp. 19–24.

McCluskey, T. L.: 2002, Knowledge engineering: Issues for the AI planning community, *The AIPS-2002 Workshop on Knowledge Engineering Tools and Techniques for AI Planning*.

McDermott, D., Ghallab, M., Howe, A., Knoblock, C., Ram, A., Veloso, M., Weld, D. and Wilkins, D.: 1998, PDDL - The planning domain definition language, *Technical report*, CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control.

Medjahed, B. and Bouguettaya, A.: 2005, A multilevel ccmposability model for Semantic Web services, *IEEE Transactions on Knowledge and Data Engineering* **17**, 954–968.

Milani, A. and Poggioni, V.: 2007, Planning in reactive environments, *Computational Intelligence* **23**(4), 439–463.

Muñoz Avila, H., Aha, D. W., Breslow, L. and Nau, D. S.: 1999, HICAP: An interactive case-based planning architecture and its application to noncombatant evacuation operations.

Muñoz Avila, H., Gupta, K., Aha, D. W. and Nau, D. S.: 2002, Knowledge-based project planning, *in* R. Dieng-Kuntz and N. Matta (eds), *Knowledge Management and Organizational Memories*, Springer, pp. 125–134.

Musliner, D. J., Durfee, E. H. and Shin, K. G.: 1991, Execution monitoring and recovery planning with time, *Conference on Artificial Intelligence Applications*, IEEE, pp. 385–388.

Myers, K. L.: 1996, Strategic advice for hierarchical planners, *International Conference on Principles of Knowledge Representation and Reasoning*, pp. 112–123.

Myers, K. L.: 2000, Planning with conflicting advice, *International Conference on Artificial Intelligence Planning Systems*, pp. 355–362. Poster paper.

Mylopoulos, J.: 1992, *Conceptual modelling and Telos*, John Wiley & Sons, Inc.

Nareyek, A., Freuder, E. C., Fourer, R., Giunchiglia, E., Goldman, R. P., Kautz, H., Rintanen, J. and Tate, A.: 2005, Constraints and AI planning, *IEEE Intelligent Systems* **20**, 62–72.

Nau, D. S.: 2007, Current trends in automated planning, *AI Magazine* **28**(4), 43–58.

Nau, D. S., Au, T. C., Ilghami, O., Kuter, U., Wu, D., Yaman, F., Muñoz Avila, H. and Murdock, J. W.: 2005, Applications of SHOP and SHOP2, *IEEE Intelligent Systems* **20**(2), 34–41.

Nau, D. S., Cao, Y., Lotem, A. and Muñoz Avila, H.: 1999, SHOP: Simple hierarchical ordered planner, *International Joint Conference on Artificial Intelligence*, IJCAI'99, pp. 968–975.

Nau, D. S., Cao, Y., Lotem, A. and Muñoz Avila, H.: 2000, SHOP and M-SHOP: Planning with ordered task decomposition, *Technical Report CS-TR-4157*, Computer Science Department, University of Maryland.

Nau, D. S., Ilghami, O., Kuter, U., Murdock, J. W., Wu, D. and Yaman, F.: 2003, SHOP2: An HTN planning system, *J. Artif. Int. Res.* **20**(1), 379–404.

Nau, D. S., Smith, S. J. and Erol, K.: 1998, Control strategies in HTN planning: Theory versus practice, *National Conference on Artificial Intelligence/Conference on Innovative applications of Artificial Intelligence*, AAAI, pp. 1127–1133.

Nejati, N., Könik, T. and Kuter, U.: 2009, A goal- and dependency-directed algorithm for learning hierarchical task networks, *International Conference on Knowledge Capture*, ACM, pp. 113–120.

Nejati, N., Langley, P. and Konik, T.: 2006, Learning hierarchical task networks by observation, *International Conference on Machine Learning*, ACM, pp. 665–672.

Neumann, J. v. and Morgenstern, O.: 1947, *Theory of games and economic behavior*, Princeton University Press.

Newell, A. and Simon, H. A.: 1976, Computer science as empirical inquiry: Symbols and search, *Commun. ACM* **19**(3), 113–126.

Nguyen, T. A., Raspitzu, A. and Aiello, M.: 2014, Ontology-based office activity recognition with applications for energy savings, *Journal of Ambient Intelligence and Humanized Computing* **5**(5), 667–681.

Nilsson, N. J.: 1980, *Principles of Artificial Intelligence*, Morgan Kaufmann Publishers Inc.

Nizamic, F., Degeler, V., Groenboom, R. and Lazovik, A.: 2012, Policy-based scheduling of cloud services, *Scalable Computing: Practice and Experience* **13**(3), 187–199.

Odersky, M., Spoon, L. and Venners, B.: 2011, *Programming in Scala: A comprehensive step-by-step guide, 2nd edition*, Artima Incorporation.

Odersky, M. and Zenger, M.: 2005, Scalable component abstractions, *SIGPLAN Not.* **40**(10), 41–57.

Ortiz, J., García-Olaya, A. and Borrajo, D.: 2013, Using activity recognition for building planning action models, *International Journal of Distributed Sensor Networks* **2013**.

Osis, J., Asnina, E. and Grave, A.: 2007, Formal computation independent model of the problem domain within the MDA, *International Conference on Information System Implementation and Modeling*, pp. 47–54.

Pagani, G. A.: 2014, *From the grid to the smart grid, topologically*, PhD thesis, Faculty of Mathematics and Natural Sciences, University of Groningen.

Pai, M., McCulloch, M., Gorman, J. D., Pai, N., Enanoria, W., Kennedy, G., Tharyan, P. and Colford, J. M.: 2004, Systematic reviews and meta-analyses: An illustrated, step-by-step guide, **17**(2), 89–95.

Paik, I. and Maruyama, D.: 2007, Automatic Web services composition using combining HTN and CSP, *IEEE International Conference on Computer and Information Technology*, pp. 206–211.

Pajares Ferrando, S. and Onaindia, E.: 2013, Context-aware multi-agent planning in intelligent environments, *Inf. Sci.* **227**, 22–42.

Papazoglou, M. P. and Georgakopoulos, D.: 2003, Introduction: Service-oriented computing, *Commun. ACM* **46**(10), 24–28.

Pattison, D. and Long, D.: 2010, Domain independent goal recognition, *Starting AI Researchers' Symposium*, STAIRS, pp. 238–250.

Pednault, E. P. D.: 1989, ADL: Exploring the middle ground between STRIPS and the situation calculus, *International Conference on Principles of Knowledge Representation and Reasoning*, pp. 324–332.

Peffers, K., Tuunanen, T., Rothenberger, M. and Chatterjee, S.: 2007, A design science research methodology for information systems research, *J. Manage. Inf. Syst.* **24**(3), 45–77.

Penberthy, J. S. and Weld, D. S.: 1992, UCPOP: A sound, complete, partial order planner for ADL, *KR*, pp. 103–114.

Petticrew, M. and Roberts, H.: 2006, *Systematic reviews in the social sciences: A practical guide*, Blackwell Publishing.

Pfender, F. and Ziegler, G. M.: 2004, Kissing numbers, sphere packings, and some unexpected proofs, *Notices-American Mathematical Society* **51**, 873–883.

Phoenix, J. and Johansson, S.: 2013, *Her*, Directed by Jonze Spike. Annapurna Pictures, Los Angeles. Film.

Plugwise: 2015. Online: accessed May 2015, http://www.plugwise.com/.

Puterman, M. L.: 1994, *Markov decision processes: Discrete stochastic dynamic programming*, 1st edn, John Wiley & Sons.

Python: 2014. Online: accessed Jan. 2014, http://www.python.org/getit/releases/3.4.0/.

Qasem, A., Heflin, J. and Muñoz-avila, H.: 2004, Efficient source discovery and service composition for ubiquitous computing environments, *Workshop on Semantic Web Technology for Mobile and Ubiquitous Applications*, ISWC'04.

Ranganathan, A. and Campbell, R. H.: 2004, Autonomic pervasive computing based on planning, *International Conference on Autonomic Computing*, ICAC'04, pp. 80–87.

Riabov, A. and Liu, Z.: 2005, Planning for stream processing systems, *National Conference on Artificial Intelligence*, AAAI, pp. 1205–1210.

Riboni, D. and Bettini, C.: 2011, COSAR: Hybrid reasoning for context-aware activity recognition, *Personal and Ubiquitous Computing* **15**(3), 271–289.

Richardson, L. and Ruby, S.: 2007, *Restful Web Services*, first edn, O'Reilly.

Robie, J., Cavicchio, R., Sinnema, R. and Wilde, E.: 2013, Restful service description language RSDL.

Rocco, M. D., Sathyakeerthy, S., Grosinger, J., Pecora, F., Saffiotti, A., Cavallo, F., Manuele, B., Limosani, R., Manzi, A., Teti, G. and Dario, P.: 2014, A planner for ambient assisted living: From high-level reasoning to low-level robot execution and back, *AAAI Spring Symposium*, pp. 10–17.

Russell, S. J. and Norvig, P.: 2003, *Artificial intelligence: A modern approach*, Pearson Education.

Sacerdoti, E. D.: 1975a, *A structure for plans and behavior*, PhD thesis, Standford University, AI Center. AAI7605794.

Sacerdoti, E. D.: 1975b, The nonlinear nature of plans, *International Joint Conference on Artificial Intelligence - Volume 1*, IJCAI'75, pp. 206–214.

Saldana, J.: 2009, *The coding manual for qualitative researchers*, Sage Publications Ltd.

Sánchez-Garzón, I., Fernández-Olivares, J. and Castillo, L.: 2011, Monitoring, repair and replanning techniques to support exception handling in HTN-based therapy planning systems, *Workshop on Artificial Intelligence in Healthcare and Biomedical Applications*.

Sánchez-Garzón, I., Fernández-Olivares, J. and Castillo, L.: 2013, An approach for representing and managing medical exceptions in care pathways based on temporal hierarchical planning techniques, *in* R. Lenz, S. Miksch, M. Peleg, M. Reichert, D. Riano and A. Teije (eds), *Process Support and Knowledge Representation in Health Care*, Vol. 7738 of *Lecture Notes in Computer Science*, Springer, pp. 168–182.

Sánchez-Garzón, I., Milla-Millán, G. and Fernández-Olivares, J.: 2012, Context-aware generation and adaptive execution of daily living care pathways, *International Conference on Ambient Assisted Living and Home Care*, Springer, pp. 362–370.

Sando, M. and Hishiyama, R.: 2011, Human-centered planning for adaptive user situation in ambient intelligence environment, *International Conference on Agents in Principle, Agents in Practice*, PRIMA'11, Springer-Verlag, pp. 520–531.

Santofimia, M. J., Fahlman, S. E., del Toro, X., Moya, F. and López, J. C.: 2011, A semantic model for actions and events in ambient intelligence, *Eng. Appl. Artif. Intell.* **24**(8), 1432–1445.

Santofimia, M. J., Fahlman, S. E., Moya, F. and López, J. C.: 2010, A common-sense planning strategy for ambient intelligence, *International Conference on Knowledge-based and Intelligent Information and Engineering Systems: Part II*, pp. 193–202.

Schattenberg, B.: 2009, *Hybrid planning and scheduling*, PhD thesis, Institute of Artificial Intelligence, Ulm University.

Shivashankar, V., Alford, R., Kuter, U. and Nau, D.: 2013, Hierarchical goal networks and goal-driven autonomy: going where AI planning meets goal reasoning, *Goal Reasoning: Papers from the ACS Workshop*, pp. 95–110.

Shivashankar, V., Kuter, U., Nau, D. S. and Alford, R.: 2012, A hierarchical goal-based formalism and algorithm for single-agent planning, *International Conference on Autonomous Agents and Multiagent Systems*, International Foundation for Autonomous Agents and Multiagent Systems, pp. 981–988.

Simek, M., Fuchs, M., Mraz, L., Moravek, P. and Botta, M.: 2011, Measurement of LowPAN network coexistence with home microwave appliances in laboratory and home environments, *International Conference on Broadband, Wireless Computing, Communication and Applications*, IEEE, pp. 292–299.

Simon, H. A.: 1996, *The sciences of the artificial*, 3 edn, MIT Press.

Simpson, R. C., Schreckenghost, D., LoPresti, E. F. and Kirsch, N.: 2006, Plans and planning in smart homes, *Designing Smart Homes*, pp. 71–84.

Simpson, R. M., McCluskey, T. L., Zhao, W., Aylett, R. S. and Doniat, C.: 2001, GIPO: An integrated graphical tool to support knowledge engineering in AI planning, *European Conference on Planning*.

Sirin, E. and Parsia, B.: 2004, Planning for Semantic Web Services, *Semantic Web Services Workshop at 3rd ISWC*.

Sirin, E., Parsia, B. and Hendler, J.: 2005, Template-based composition of Semantic Web services, *AAAI Fall Symposium on Agents and the Semantic Web*, pp. 85–92.

Sirin, E., Parsia, B., Wu, D., Hendler, J. and Nau, D. S.: 2004, HTN planning for Web service composition using SHOP2, *Web Semantic* **1**, 377–396.

Smidts, C., Mutha, C., Rodríguez, M. and Gerber, M. J.: 2014, Software testing with an operational profile: Op definition, *ACM Comput. Surv.* **46**(3), 39:1–39:39.

Smith, D. E., Frank, J. and Jónsson, A. K.: 2000, Bridging the gap between planning and scheduling, *Knowl. Eng. Rev.* **15**(1), 47–83.

Smith, D. E. and Weld, D. S.: 1999, Temporal planning with mutual exclusion reasoning, *International Joint Conference on Artificial Intelligence*, IJCAI'99, pp. 326–337.

Smith, S. J. J., Hebbar, K., Nau, D. S. and Minis, I.: 1997, Integrating electrical and mechanical design and process planning.

Sohrabi, S.: 2013, *Customizing the composition of Web services and beyond*, PhD thesis, Depart. of Computer Science, Univ. of Toronto.

Sohrabi, S., Baier, J. A. and McIlraith, S. A.: 2008, HTN planning with quantitative preferences via heuristic search, *Workshop on Oversubscribed Planning and Scheduling at ICAPS*.

Sohrabi, S., Baier, J. A. and McIlraith, S. A.: 2009, HTN planning with preferences, *International Joint Conference on Artifical Intelligence*, IJCAI'09, pp. 1790–1797.

Sohrabi, S. and Mcilraith, S. A.: 2008, On planning with preferences in HTN, *International Workshop on Non-Monotonic Reasoning*, pp. 241–248.

Sohrabi, S. and Mcilraith, S. A.: 2009, Optimizing Web Service composition while enforcing regulations, *International Semantic Web Conference*, ISWC '09, pp. 601–617.

Sohrabi, S. and McIlraith, S. A.: 2010, Preference-based Web service composition: A middle ground between execution and search, *International Semantic web Conference on The semantic web - Volume Part I*, ISWC'10, pp. 713–729.

Sohrabi, S., Prokoshyna, N. and Mcilraith, S. A.: 2006, Web service composition via generic procedures and customizing user preferences, *International Semantic Web Conference*, ISWC'06, pp. 597–611.

Sohrabi, S., Udrea, O. and Riabov, A.: 2013, HTN planning for the composition of stream processing applications, *International Conference on Automated Planning and Scheduling*, ICAPS'13, pp. 443–451.

Song, S. and Lee, S.-W.: 2013, A goal-driven approach for adaptive service composition using planning, *Mathematical and Computer Modelling* **58**(1-2), 261–273.

Stavropoulos, T. G., Vrakas, D. and Vlahavas, I.: 2011, A survey of service composition in ambient intelligence environments, *Artificial Intelligence Review* pp. 1–24.

Stefik, M.: 1981, Planning with constraints (MOLGEN: Part 1), *Artif. Intell.* **16**(2), 111–140.

Stillman, J., Arthur, R. and Deitsch, A.: 1993, Tachyon: A constraint-based temporal reasoning model and its implementation, *SIGART Bull.* **4**(3), 1–4.

Sutcliffe, A.: 2003, Scenario-based requirements engineering, *International Requirements Engineering Conference*, pp. 320–329.

Taqqali, W. M. and Abdulaziz, N.: 2010, Smart grid and demand response technology, *International Energy Conference and Exhibition*, IEEE, pp. 710–715.

Tate, A.: 1976, Project planning using a hierarchic non-linear planner, *Technical Report 25*, Department of Artificial Intelligence, University of Edinburgh.

Tate, A.: 1977, Generating project networks, *International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'77, pp. 888–893.

Tate, A.: 1994, Key concepts in the O-Plan2 knowledge based plan representation, *UK Planning and Scheduling*.

Tate, A.: 2007, Planning and doing things, *AISB Quarterly* pp. 7–8.

Tate, A. and Dalton, J.: 2003, O-Plan: A common Lisp planning Web service, *International Lisp Conference*.

Tate, A., Dalton, J. and Levine, J.: 1998, Generation of multiple qualitatively different plan options, *International Conference on AI Planning Systems*, AAAI, pp. 27–35.

Tate, A., Dalton, J. and Levine, J.: 2000, O-Plan: A Web-based AI planning agent, *National Conference on Artificial Intelligence/Conference on Innovative Applications of Artificial Intelligence*, AAAI, pp. 1131–1132.

Tate, A., Drabble, B. and Dalton, J.: 1994, The use of condition types to restrict search in an AI planner, *National Conference on Artificial Intelligence - Volume 2*, AAAI, pp. 1129–1134.

Tate, A., Drabble, B. and Dalton, J.: 1996, O-Plan: A knowledge-based planner and its application to logistics, *ARPI*, pp. 259–266.

Tate, A., Drabble, B. and Kirby, R.: 1994, O-Plan2: An open architecture for command, planning and control, *Intelligent Scheduling*, pp. 213–239.

Tate, A. and Lesley, D.: 1982, A retrospective on the 'Planning: A Joint AI/OR Approach' project, *Technical Report Working paper 125*, Department of Artificial Intelligence, University of Edinburgh.

Tate, A., Levine, J., Jarvis, P. and Dalton, J.: 2000, Using AI planning technology for army small unit operations, *Artificial Intelligence Planning and Scheduling Systems Conference*, pp. 379–386. Poster paper.

Thalheim, B.: 2010, Towards a theory of conceptual modelling, *Journal of Universal Computer Science* **16**(20), 3102–3137.

Thalheim, B.: 2011, The art of conceptual modelling, *in* J. Henno, Y. Kiyoki, T. Tokuda, H. Jaakkola and N. Yoshida (eds), *European-Japanese Conference on Information Modelling and Knowledge Bases*, Vol. 237 of *Frontiers in Artificial Intelligence and Applications*, pp. 149–168.

Tsuneto, R., Erol, K., Hendler, J. and Nau, D. S.: 1996, Commitment strategies in hierarchical task network planning, *National Conference on Artificial Intelligence - Volume 1*, AAAI'96, AAAI, pp. 536–542.

Tsuneto, R., Hendler, J. and Nau, D. S.: 1998, Analyzing external conditions to improve the efficiency of HTN planning, *National Conference on Artificial Intelligence/-Conference on Innovative Applications of Artificial Intelligence*, AAAI, pp. 913–920.

Urbieta, A., Barrutieta, G., Parra, J. and Uribarren, A.: 2008, A survey of dynamic service composition approaches for ambient systems, *Ambi-Sys Workshop on Software Organisation and MonIToring of Ambient Systems*, SOMITAS'08, pp. 1:1–1:8.

Uszok, A., Bradshaw, J. M., Jeffers, R., Tate, A. and Dalton, J.: 2004, Applying KAoS services to ensure policy compliance for Semantic Web services workflow composition and enactment, *International Semantic Web Conference*, pp. 425–440.

Vaquero, L. M., Rodero-Merino, L., Caceres, J. and Lindner, M.: 2008, A break in the clouds: Towards a cloud definition, *SIGCOMM Comput. Commun. Rev.* **39**(1), 50–55.

Veloso, M. M., Pollack, M. E. and Cox, M. T.: 1998, Rationale-based monitoring for planning in dynamic environments, *International Conference on Artificial Intelligence Planning Systems*, pp. 171–180.

Vukovic, M., Kotsovinos, E. and Robinson, P.: 2007, An architecture for rapid, on-demand service composition, *Service Oriented Computing and Applications* **1**(4), 197–212.

Walsh, J. D., Bordeleau, F. and Selic, B.: 2007, Domain analysis of dynamic system reconfiguration, *Software & Systems Modeling* **6**(4), 355–380.

Weerdt, M. d. and Clement, B.: 2009, Introduction to planning in multiagent systems, *Multiagent Grid Syst.* **5**(4), 345–355.

Weiser, M.: 1999, The computer for the 21st century, *SIGMOBILE Mob. Comput. Commun. Rev.* **3**(3), 3–11.

Weld, D. S.: 1994, An introduction to least commitment planning, *AI Magazine* **15**(4), 27–61.

Weser, M., Off, D. and Zhang, J.: 2010, HTN robot planning in partially observable dynamic environments, *International Conference on Robotics and Automation*, pp. 1505–1510.

Whitehead, A. N.: 2010, *Process and reality*, Simon and Schuster. First edition 1929.

Wichansky, A. M.: 2000, Usability testing in 2000 and beyond, *Ergonomics* **43**(7), 998–1006.

Wilkins, D. and Desimone, R. V.: 1992, Applying an AI planner to military operations planning, *Intelligent Scheduling*, pp. 685–709.

Wilkins, D. E.: 1988, *Practical planning: Extending the classical AI planning paradigm*, Morgan Kaufmann Publishers Inc.

Wilkins, D. E.: 1991, Can AI planners solve practical problems?, *Comput. Intell.* **6**, 232–246.

Wilkins, D. E. and Desjardins, M.: 2001, A call for knowledge-based planning, *AI Magazine* **22**(1), 99–115.

Wilkins, D. E., Lee, T. and Berry, P.: 2003, Interactive execution monitoring of agent teams, *Journal of Artificial Intelligence Research* **18**, 217–261.

Wiser, R., Barbose, G. and Peterman, C.: 2009, Tracking the sun: The installed cost of photovoltaics in the U.S. from 1998-2007, *Technical report*, Lawrence Berkeley National Laboratory.

Wooldridge, M.: 2009, *An introduction to multi-agent systems*, Wiley Publishing.

Wu, D., Parsia, B., Sirin, E., Hendler, J. and Nau, D. S.: 2003, Automating DAML-S Web services composition using SHOP2, *International Semantic Web Conference*, ISWC'03, pp. 195–210.

Wyatt, D.: 2013, *Akka Concurrency*, Artima Incorporation.

Yaman, F. and Nau, D. S.: 2002, Timeline: An HTN planner that can reason about time, *AIPS'02 Workshop on Planning for Temporal Domains*, pp. 75–81.

Yang, Q.: 1992, A theory of conflict resolution in planning, *Artif. Intell.* **58**(1), 361–392.

Yordanova, K.: 2011, Modelling human behaviour using partial order planning based on atomic action templates, *International Conference on Intelligent Environments*, pp. 338–341.

Zhuo, H. H., Muñoz Avila, H. and Yang, Q.: 2014, Learning hierarchical task network domains from partially observed plan traces, *Artificial Intelligence* **212**(0), 134–157.

Samenvatting

Ubiquitous computing wordt in toenemende mate gerealiseerd door het hebben van diverse geïntegreerde apparatuur en alom aanwezige toepassingen in onze omgeving. Zo is elke kamer in je huis is voorzien van diverse huishoudelijke apparaten, zoals een TV in de woonkamer. Ook kan het huis worden verrijkt met een tal van onopvallende apparatuur, zoals radio-frequency identification tags, temperatuur en gas-lekkage sensoren, actuatoren om lampen te bedienen, *etc.* Deze apparaten bieden verschillende soorten van informatie en middelen ter bediening via een breed scala aan communicatie en integratie technologieën, doorgaans zichtbaar door een tal van services. Een service is een abstractie van een autonoom software component vanuit zijn implementatiedetails. Een lamp heeft bijvoorbeeld services voor zowel het aftasten en wijzigen van zijn toestand. De echte voordelen van dergelijke verrijkte en geabstraheerde omgevingen komen naar voren wanneer de focus ligt op de afstemming van services ten behoeve van het grotere geheel, zoals het verbeteren van; de ervaring en kwaliteit van leven van de mens, energie en monetaire besparingen, of veiligheid.

Service afstemming behelst het selecteren en combineren van services ten behoeve van een bepaald verzoek. Gezien het feit dat diensten snel oplopen, bijvoorbeeld doordat nieuwe apparaten worden toegevoegd in het huis, hun beschikbaarheid constant verandert, en verzoeken aangepast kunnen worden, wordt de afstemming van services een complex proces dat autonoom en intelligent moet worden uitgevoerd. Het domein van Artificial Intelligence planning kan middelen verstrekken voor het geautomatiseerd en dynamisch afstemmen van services omdat planning betrekking heeft op het selecteren en combineren van acties door de overweging van hun resultaten om een gegeven doelstelling automatisch te realiseren. Daarmee komen acties overeen met services en doelstellingen met verzoeken. De fundamentele en evidente overeenkomst tussen planning en ubiquitous computing omgevin-

gen wordt benut in een aantal bestaande studies. Wat schijnbaar minder voor de hand ligt is de hoe ubiquitous computing gerelateerd is aan planning buiten services en gebruikersverzoeken. Daarom hebben we een conceptueel model van het planningsdomein voor ubiquitous computing gemaakt die de entiteiten waaruit het domein bestaat, en de relaties tussen hen, kenmerkt. Gezien dat het model gebaseerd is op al bestaande literatuur wordt de aansluiting bij bestaande benaderingen gewaarborgd. Het model dient ter bevordering van een meer efficiënter ontwerp en ontwikkeling van toekomstige systemen in ubiquitous computing. Verder het over het algemeen niet duidelijk waaruit een planning taak bij ubiquitous computing kan bestaan en wat de complexiteit van het oplossen van een dergelijke taak is. We hebben daarom een algemeen planning domein voor ubiquitous computing gedefinieerd, en verschaffen initiële complexiteitsresultaten voor het oplossen van planningsproblemen binnen dat domein.

Vanuit het aanbod aan planning technieken richten we ons op Hierarchical Task Network (HTN) planning, hoofdzakelijk vanwege de bijkomende omvangrijke domein kennis en de voordelen die daaruit voortvloeien. Hoewel HTN planning al lang bestaat en op grote schaal wordt gebruikt, wordt het gekenmerkt door controverse en het ontbreken van algemeen begrip. Deze situatie kan niet probleemloos worden verklaard, aangezien de huidige literatuur over HTN planning, weliswaar omvangrijk, weinig tot niets rapporteert over de problematiek. We verhelpen dit door het verzamelen van informatie over de formele modellen, concepten en eigenschappen van bestaande planners en studies. Onze bijdrage bestaat uit categorisaties van het onderzoeksveld, en ophelderingen van de vele misvattingen geassocieerd aan deze techniek. Verder verbeteren we HTN planning met een aantal nieuwe eigenschappen: Toepassingen die nuttig zijn voor domeinen met grote wint of verliest van middelen, zoals energie in ubiquitous computing, automatic phantomisation om een domein ontwerper te ontlasten met het identificeren en coderen van een aantal specifieke situaties, en service-oriëntatie om eenvoudigere integratie van planners in service-oriented systemen mogelijk te maken. Ook ontwerpen en ontwikkelen we een HTN planning systeem, genaamd Scalable Hierarchical (**SH**) planner, die een goed gedefinieerde syntaxis accepteert, en kan worden geïntegreerd in een breed scala aan grote en gedistribueerde systemen. We gebruiken de planner in de domeinen van ubiquitous computing en cloud computing.

Een van onze hoofddoelen is om een verband tussen ubiquitous computing en HTN planning vast te stellen, en om HTN planning te gebruiken voor een praktische toepassing. We stellen daarom een juiste overeenkomst tussen een ubiquitous computing probleem en een HTN planning probleem voor. Het gevolg hiervan is het feit dat het service plan berekend voor het HTN planning probleem inderdaad een oplossing voor het onderliggende ubiquitous computing probleem is. Daar-

naast verbeteren we deze benadering met een functie, genaamd orchestration, om de vraag, hoe om te gaan met tegenstrijdigheden tijdens executie, te beantwoorden. Om de haalbaarheid van de benadering te evalueren, ontwerpen we een systeemarchitectuur en implementeren we een systeemprototype. We implementeren en passen het prototype toe op het restaurant in de Bernoulliborg van de universiteit van Groningen. De resultaten duiden erop dat HTN planning met zijn rijke domein kennis een bruikbare en effectieve techniek voor de afstemming van alom geïntegreerde services is. De effectiviteit wordt aangetoond door energie en monetaire besparingen. De haalbaarheid van de benadering wordt bevestigd door middel van gebruiksvriendelijkheidsevaluaties en prestatiebeoordelingen. Uit de gebruiksvriendelijkheidsevaluaties blijkt dat de meerderheid van de deelnemers vindt dat het systeem nuttig en effectief is. Uit de prestatiebeoordelingen blijkt dat de benodigde tijd voor het berekenen van plannen in realistische situaties in de orde van milliseconden ligt, en in extreme gevallen enkele seconden bedraagt.

We richten ons mede op het onderzoeken van de problematiek rond het optimaliseren van de monetaire- en energiekosten voor ubiquitous computing omgevingen aangesloten op het smart grid. We stellen een gecentraliseerde benadering voor op basis van het plannen van diensten ter besturing van apparaten binnen dergelijke omgevingen. Om de haalbaarheid van de benadering te evalueren, ontwerpen en realiseren we een tweede systeemprototype, dat we inzetten in kantoren op de vijfde verdieping van de Bernoulliborg. De resultaten tonen aan dat onze benadering een effectieve manier biedt om apparaat services af te stemmen terwijl het gebruik van energie in de piekuren wordt afgewogen. De effectiviteit van het systeemprototype komt tot uitdrukking in de besparingen van tot wel 50% aan geld en 15% aan energie.

Tot slot streven we ernaar om te weten hoe het proces van het samenstellen van "ready to use" cloud-applicaties kan worden geautomatiseerd met behulp van HTN planning. Derhalve komen we tot een overeenkomst tussen een deployment probleem en een HTN planning probleem. Om de haalbaarheid van de benadering te evalueren, coderen we cloud services in HTN domeinkennis en gebruiken de **SH** planner om deployment-gebaseerde HTN planning problemen op te lossen. We testen **SH** op een reeks problemen waarvan de prestatieresultaten laten zien dat HTN planning met zijn rijke domeinkennis in staat is om de deployment-gebaseerde HTN planning problemen op te lossen in een kwestie van enkele seconden.

Kortom, we dragen bij met een intelligent systeem dat alom aanwezig kan zijn. Het systeem stemt de alom vertegenwoordigde services automatisch en dynamisch af, maar kan ook rekening houden met de prijs van de energie die men moet betalen voor de afgestemde apparaten. Dit impliceert een betere kwaliteit van leven van

mensen in termen van comfort, welzijn en economie. Desalniettemin kan men hun eisen en behoeften voor hun huis uiten, en kan het systeem zichzelf in korte tijd automatisch upgraden met de nodige cloud diensten.